NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

QUALITY NETWORK LOAD INFORMATION IMPROVES PERFORMANCE OF ADAPTIVE APPLICATIONS

by

John P Kresho

September 1997

Thesis Advisor: Debra Hensgen
Second Reader: Taylor Kidd

Approved for public release; Distribution is unlimited.

DTIC QUALITY INSPECTED

19980209 081

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE September 1997	3. REPORT TYPE AND DATES COVERED Master's Thesis	
4. TITLE AND SUBTITLE Quality Network Load Information Improves Performance of Adaptive Applications		5. FUNDING NUMBERS	
6. AUTHOR(S) Kresho, John P			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey CA 93943-5000		8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSORING/MONITORING AGENCY REPORT NUMBER	

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited	12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

The Joint Task Force Reference Architecture requires a Comms Server to aid client applications in adapting to changing network loads by apprising them of current and expected loads. The current Comms Server implementation estimates the network load by sending various sized packets and reporting raw performance statistics to the client. This implementation presents three problems: (1) clients interpret the statistics autonomously, (2) statistics are inaccurate due to the instantaneous collection procedure, and (3) clients also require the state of other resources to make informed decisions concerning adaptation. Development of a new Comms Server design, which solves these problems, is needed.

This thesis develops a new Comms Server design and determines, through simulation, whether providing a more accurate estimate of the load could permit users of adaptive applications to obtain better performance. Simulations were run using many different situational parameters. Both the average size of the data successfully transmitted, and whether an application met its deadline, were recorded.

The results of these simulations show that clients of the existing Comms Server perform much better because they adapt, but in some cases 14% to 30% of the messages do not arrive by their deadline. However, a better design that more accurately estimates loads could deliver at least 96% of the messages on time.

14. SUBJECT TERMS Software Architectures, Adaptive Applications, Resource Monitoring, Network Simulation			15. NUMBER OF PAGES 185
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

QUALITY NETWORK LOAD INFORMATION IMPROVES PERFORMANCE OF ADAPTIVE APPLICATIONS

John P Kresho
Captain, United States Marine Corps
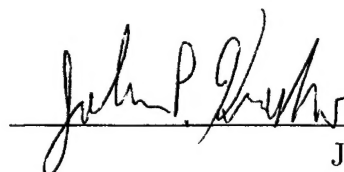B.S., Cornell Univerity, 1991

Submitted in partial fulfillment of the
requirements for the degree of
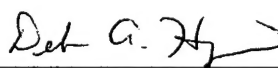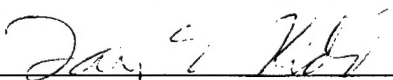
MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

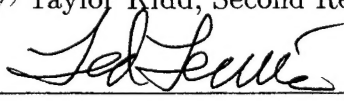## NAVAL POSTGRADUATE SCHOOL
September 1997

Author: _____
John P Kresho

Approved by: _____
Debra Hensgen, Thesis Advisor

_____
Taylor Kidd, Second Reader

_____
Ted Lewis, Chairman
Department of Computer Science

ABSTRACT

The Joint Task Force Reference Architecture requires a Comms Server to aid client applications in adapting to changing network loads by apprising them of current and expected loads. The current Comms Server implementation estimates the network load by sending various sized packets and reporting raw performance statistics to the client. This implementation presents three problems: (1) clients interpret the statistics autonomously, (2) statistics are inaccurate due to the instantaneous collection procedure, and (3) clients also require the state of other resources to make informed decisions concerning adaptation. Development of a new Comms Server design, which solves these problems, is needed.

This thesis develops a new Comms Server design and determines, through simulation, whether providing a more accurate estimate of the load could permit users of adaptive applications to obtain better performance. Simulations were run using many different situational parameters. Both the average size of the data successfully transmitted, and whether an application met its deadline, were recorded.

The results of these simulations show that clients of the existing Comms Server perform much better because they adapt, but in some cases 14% to 30% of the messages do not arrive by their deadline. However, a better design that more accurately estimates loads could deliver at least 96% of the messages on time.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

First, I thank most of all my loving and patient wife Deborah and son Charles. Thank you to Debra Hensgen and Taylor Kidd who opened my eyes to architecture. I also want to thank Joel Levin, Mike Dean and Chuck Blake from BBN for answering all of my questions.

I.   INTRODUCTION

This thesis investigates the software infrastructure necessary to support good performance for a video teleconferencing (VTC) setup application in a Joint Task Force environment. It builds upon the architectural components that are currently part of the Joint Task Force - Advanced Technology Demonstration (JTF ATD) program. In particular, it investigates better ways to allocate shared communication resources. Additionally it examines whether or not, for such communication-intensive applications, other resources must also be carefully managed. This thesis proposes a server-client solution that uses a set of servers to keep track of the current resource loads. Those servers are updated by library stubs that are linked with each client. In addition to investigating and reporting on various resource allocation policies that we examined through the use of a simulation, we also document the ease with which we were able to use the JTF ATD reference architecture (Figure 1) to build a prototype VTC setup system.

A.   BACKGROUND

When a crisis occurs in the world that threatens the interests of the United States, the U.S. military forces are called into service. In order to quickly respond to the situation, a Crisis Support Team will be formed which is composed of different U.S. services. The commander of this Joint Task Force (JTF) must communicate with each component of this force. These components may be geographically dispersed, and further, they often have hardware and software from different vendors, which could complicate, rather than facilitate, communication.

In order to ensure that the JTF commander is provided a working command and control (C2) system, there must be a set of standards to guarantee that each service's and vendor's equipment is compatible with each other. In the past, each service has independently specified its own requirements. For example, when the Navy

1
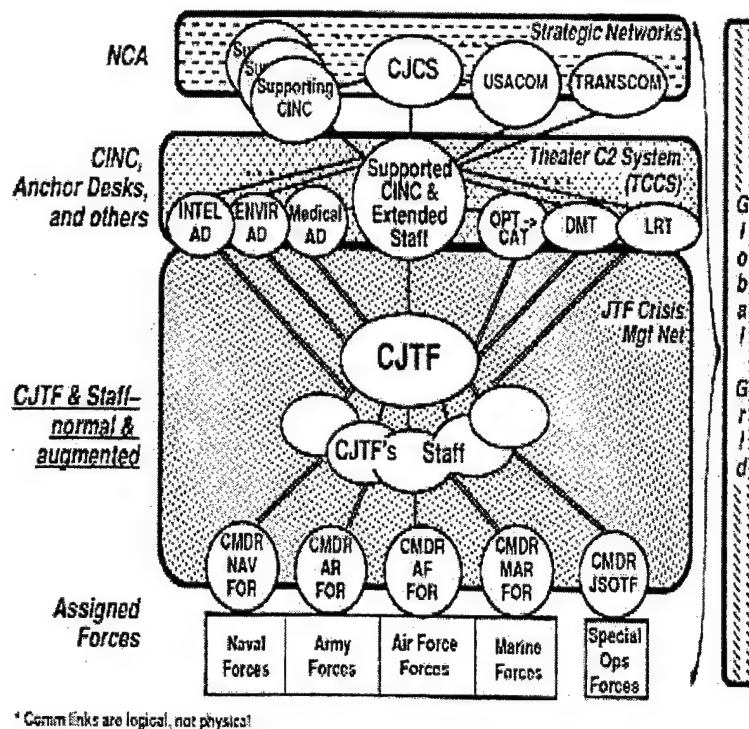
Figure 1. JTF context for employment. The JTF staff communicates with a variety of other entities, making use of several communication networks. Note that the lines connecting various entities are meant to indicate the most frequent interactions; they are not meant to indicate dedicated communication links. In general, all the involved individuals are networked together, with the possibility of communication as needed (From [Ref. 1]).

built a new ship, it would ensure that its communications equipment was compatible with the rest of its ships and its shore installations. The Army was doing the same thing on the ground. It built communication systems that met their needs to allow its infantry to talk to the artillery component. However, when the Army and the Navy had to cooperate to solve a crisis, their equipment was incompatible and much of their time was wasted in order to solve this inter-service communication problem. In the meantime, no progress was made towards resolving the crisis.

Many of the problems described above stem from the fact that each service's communications equipment was built by a different contractor, for a specific purpose.

This drastically reduces the probability of any kind of standardization of hardware or software. Additionally, by the time that a contractor meets the needs of a particular specification, the technology is outdated, and, consequently, our services are operating behind the leading edge of technology.

Due to the shrinking of the Armed Forces along with its budget, Joint Operations are becoming common place. In order to overcome these shortcomings of incompatible and outdated systems, some sort of standardization is needed. In 1994, the Defense Advanced Research Projects Agency (DARPA) contracted Teknowledge Federal Systems to write a Joint Task Force Architecture Specification. The goal was to provide an initial architecture that would allow for rapid integration of new C2 hardware and software. This new software architecture will allow for the use of leading technology and present a flexible and evolvable environment, providing the JTF commander an anytime/anywhere information support system.

The Teknowledge Federal Systems' JTF Architecture Specification resulted in the JTF ATD [Ref. 2]. Many different sites around the world, with different hardware architectures, are connected using different communication networks. These sites use a common set of software services that are specified by the JTF Reference Architecture. This set of services is chosen to allow other developers to easily build future C2 applications in a short time, such as within a few days or weeks.

B. MOTIVATION

Currently, there are few applications that have been developed that fully exercise the JTF ATD architecture. A basic architecture implementation is in place, and now the time has come to test the flexibility and performance of this architecture. In addition to performance questions, this thesis seeks to begin to answer other questions concerning this reference architecture, such as those listed below.

- How long will it take a developer of an application to become familiar with the structure of the generic services provided?

3

- Does the JTF Reference Architecture facilitate the reuse of its existing objects and services?

- Does the JTF ATD Communications Server provide sufficient information to applications, allowing them to maximize performance, given the status of the network?

- In addition to the network resource, should the JTF ATD regulate other machine resources for all applications, even though the majority of the initial ones will be I/O-intensive?

One of the major concerns of any commander is how coordination, communication, and control will be handled. One of the most effective ways of communicating is face-to-face. This form of communication allows the commander to use hand gestures, visual objects, and facial expressions to ensure that his fellow planners completely understand his intent. Since many of the components of the JTF may be geographically dispersed, the latest technology in video teleconferencing must be used to meet the commander's communication and control requirements.

However, the *coordination* of such a meeting is not currently facilitated by the JTF ATD software. Even when using this software, setting up a video teleconferencing session requires many phone calls to ensure that all locations are synchronized and will be available for the video conference. In addition to having all participants present, it is vital that each site have the required materials (i.e., applications, slides, text files) to actively participate in the video conference.

C.   PURPOSE

In order to overcome the burden inherent in the current manual setup of the video teleconference session, and to ensure that an application can succeed even when network traffic is bursty, we developed a prototype application that automatically set up a video teleconferencing session. The teleconferencing setup application notified participants of any problems with the delivery of files or applications. This application extensively uses the services provided by the Communications Server, Worklow Server,

and Trigger Server, as well as other underlying servers provided by the JTF Reference Architecture.

Our VTC setup application must be **Network Aware**. By this we mean that it must adapt to the current network availability. In order to do this, we permit users to specify alternative files and/or applications. For example, a user might prefer that an entire video sequence be broadcast to all participants, but if that user's bandwidth to certain participants is (currently) low, the user might specify that some still photos, or even a textual description file, might be used instead.

Providing feedback to the designers of the JTF ATD Reference Architecture is vital to its success. While implementing this application, the servers mentioned above were tested in order to provide insight on how well they meet the clients' needs, and what things might better be done differently.

Our initial tests indicated that the existing Communications Server Architecture by itself will likely not provide sufficient support for network adaptability. We therefore propose an alternate software architecture. We also compare, via simulation, the performance of alternate architectures with that provided by the Communications Server.

D.   ORGANIZATION

Chapter II gives an overview of the entire JTF Reference Architecture. Chapter III describes the VTC setup application and the JTF ATD servers that we used. Chapter IV presents our initial results from testing the Communications Server. Chapter V details the simulation software we built and discusses the results from these simulations. Chapter VI formalizes the mathematical problem that must be solved to allow adapt-

ive applications to receive the best quality of service, and Chapter VII discusses a proposed architecture to solve this problem. Chapter VIII summarizes our findings and suggests future work that builds upon our findings.

II. OVERVIEW OF THE JTF REFERENCE ARCHITECTURE

The goal of a reference architecture is to reduce the time associated with application integration. The reference architecture specifies the characteristics from which a system will be built. Knowing these characteristics allows a programmer to develop applications quickly by reusing existing objects. For example, a particular reference architecture might specify that only certain Graphical User Interface (GUI) packages may be used. A programmer who knows this characteristic can reuse many of the existing windows and buttons developed for the architecture. A well defined reference architecture will constrain the development and purchasing of hardware and software meant to be integrated with a particular system, while still allowing the system to make use of the most current COTS and GOTS software and hardware. In the case of the JTF ATD, its reference architecture will ensure that all of the Armed Services purchase and produce compatible equipment that will work together seamlessly during a crisis.

The JTF Reference Architecture is broken up into four distinct layers [Ref. 2]. Each layer uses the functionality of the layers below it. Figure 2 shows a conceptual representation of these levels in the JTF Reference Architecture. Beginning at the top, the following is a list of these levels:

1. The User Environment,

2. Applications that aid the Commander of the JTF and his staff,

3. Generic Servers that provide various C2 and generic access to computing resources, and

4. The base-level infrastructure that provides the distributed computing environment.

7

User Environments		
Task Force Staff Process Management	Situation Assessment & Planning	Coordination, Communication, & Control
Generic Servers: Data, Models, Messages, Comm., Maps, Plans, Situations, Webs		
Infrastructure: Collaborative Object Management, Communications, & Computing Infrastructure		
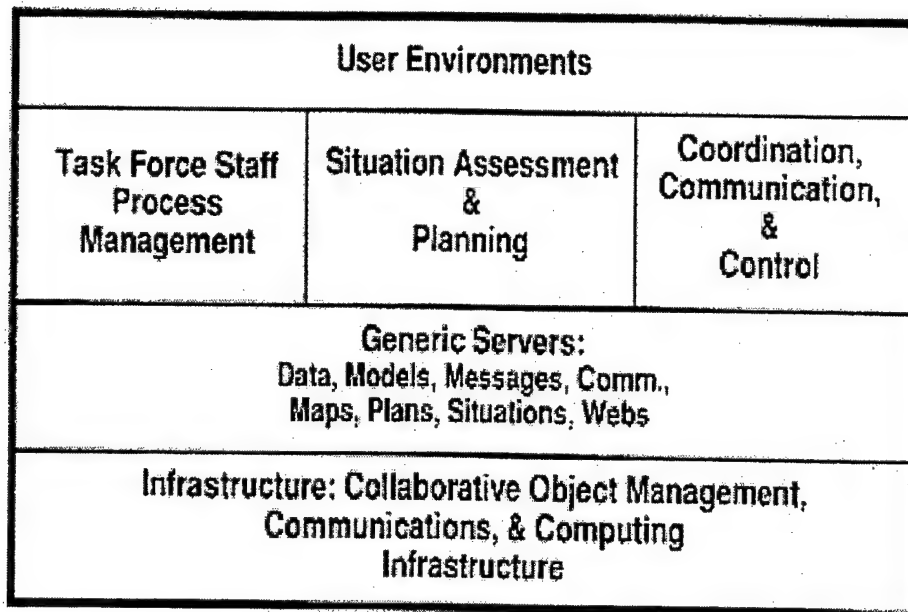
JTF staff interact with systems through environments (1)
to access applications that support tasks (2)
using provided services (3)
over distributed collaborative networks (4)

Figure 2. Four Levels of the JTF Reference Architecture (From [Ref. 1]).

A.   USER ENVIRONMENT

The User Environment, an example of which is shown in Figure 3, specifies a common interface to be used throughout all of the JTF applications. With a standard look and feel for interfaces, such as dialog bars, buttons, and scroll bars, applications built using the JTF Reference Architecture will behave similarly, thereby minimizing the ramp up time necessary for effective execution of new applications.

In order to help maintain a standard Human Computer Interface (HCI), the JTF ATD Reference Architecture further constrains the application developers. In its evaluation of current technologies, five GUI tools have been adopted. They are Motif/X11, Tcl/Tk, HTML++, Java applets for HTML++ Browsers, and Java Viewers. Only applications developed using one of these five tools can be JTF-compliant.

8
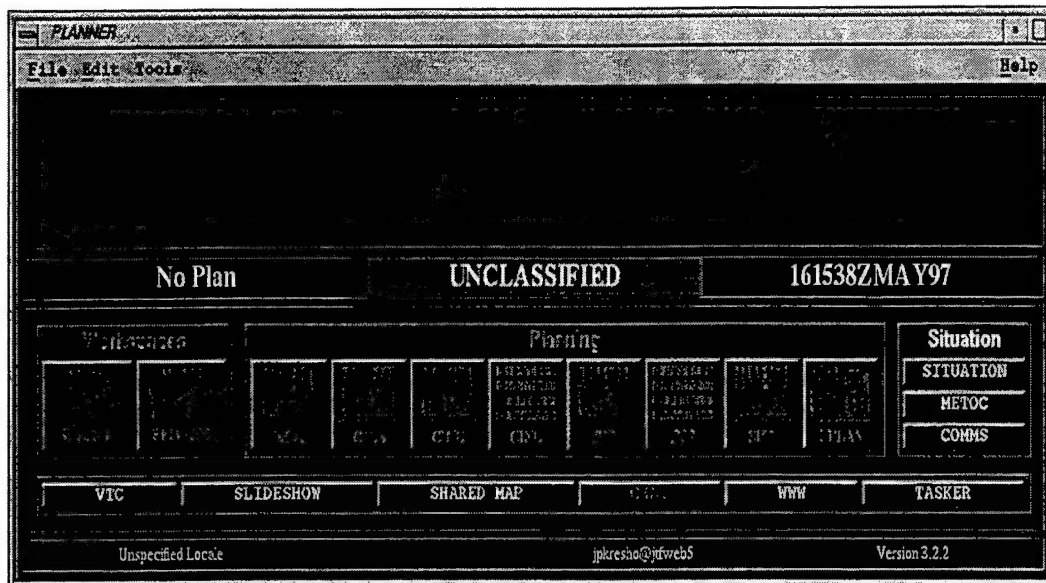
Figure 3. JTF ATD Reference Architecture User Environment Example.

B. PRE-SPECIFIED APPLICATIONS LAYER

The applications provided by this layer directly support the decision making of the JTF staff. Applications specified in this layer manage the resources needed to aid decision making and the implementation of those decisions. For example, a commander may have an application that gathers important logistical data from his units. Based on this data, another application will quickly distribute his plans to his subordinates.

Appropriately, this layer is divided into three sublayers: Task Force Staff Process Management; Situation Assessment and Planning; and Coordination, Communication, and Control(Figure 2). Currently, the majority of the functionality of this layer is performed by humans, but the JTF Reference Architecture specifies these functionalities to give developers implementation guidelines for automating them.

1. Task Force Staff Process Management

The Commander of the JTF will not know where and with whom the next crisis will be fought. He will not know what resources will be available, or of what quality those resources will be. The Task Forces Process Management module has eleven

9

principal functions that are used to help the JTF staff quickly assess new situations. These functions aid the user in determining resources, specifying policies, specifying tasks, organizing and delegating tasks, scheduling tasks, allocating resources, performing tasks, monitoring the developing situation, process replanning, assessing task performance, and improving task performance.

2. The Situation Assessment and Planning Module

The second module of the Application layer is the Situation Assessment and Planning Module. This module contains functions to support the following nine general categories of the decision-making process of the JTF staff: situation assessment, scenario generation, plan generation, plan evaluation, plan selection, plan analysis, plan monitoring, replanning, and cut-over planning.

3. The Coordination, Communication, and Control Module

The final module of the second layer is the Coordination, Communication, and Control Module. This module enables the staff of the JTF to pass information to each other and allows for valuable input from other organizations. There are five general functionalities to support information exchange:

- Generate briefings, reports, orders, and requests;
- Convey briefings, reports, orders, and requests;
- Receive briefings, reports, orders, and requests;
- Assess received briefings, reports, orders, and requests; and
- Disseminate received briefings, reports, orders, and requests.

Our video teleconference setup application resides in this layer.

C.   GENERIC SERVICES

The next layer, the Generic Services, provides the Applications Layer with the common services that are used by more than one JTF application. This layer, along

with the one below it, facilitates communication between different applications in the layers above.

Servers in this layer provide such services as the handling of maps, the delivery of messages, and the storage and retrieval of data. Additionally, it defines a standard representation for each of these data objects. In order to specify the proper handling of each class of data types, several servers are provided by the JTF architecture [Ref. 2].

Communications Server. Brokers the limited communications bandwidth to consumer applications via contracts for latency and quality of service. It charges a price that balances supply and demand.

Data Server. Employs a common object-oriented C2 schema that provides its clients with periodically updated query-based views of distributed, heterogeneous databases.

Web Server. Provides its clients with the means to construct, distribute, view, edit, and replicate node-link structures that incorporate objects of arbitrary types. For example, Figure 4 shows a sequence of events that would effect a web containing several different objects. These objects all relate to the same situation, and, when one object changes, the web updates itself. For instance, when III MEF's logistical database is updated to show that it has 5 less trucks available due to maintenance problems, the USCINPAC's database is automatically updated, along with both tactical maps showing that these trucks are no longer in their respective positions. (Note: "Web" does not refer to the Internet).

Situation Server. Enables its clients to develop interpretations or "pictures" of the battle space that incorporate objects, aggregates, inferences, and predictions. All of these data are indexed over space, time, and their assumed context.

Plan Server. Enables a group of geographically distributed planners to jointly hypothesize, evaluate, and disseminate alternative courses of actions (COAs).

Model Server. Initializes and executes simulations that assess COAs in the context of various assumed situations.

Map Server. Coordinates the maps that pertain to the current situation for which the JTF staff is planning. When a change occurs in the plan or situation, the Map Server automatically updates the appropriate maps.

Message Server. Provides user applications with a single interface for accessing, composing, editing, routing, and dispatching military messages.

Workflow Server. Provides its clients with objects that contain the data to produce mission type orders. This server tracks the flow of information to and from all participants of a particular mission order.

Socket Trigger Server. Provides its client with the capability to asynchronously await an event. For example, while typing a mission plan, a user can receive an email message. The email application contains a trigger that will notify the user by producing a popup message box.
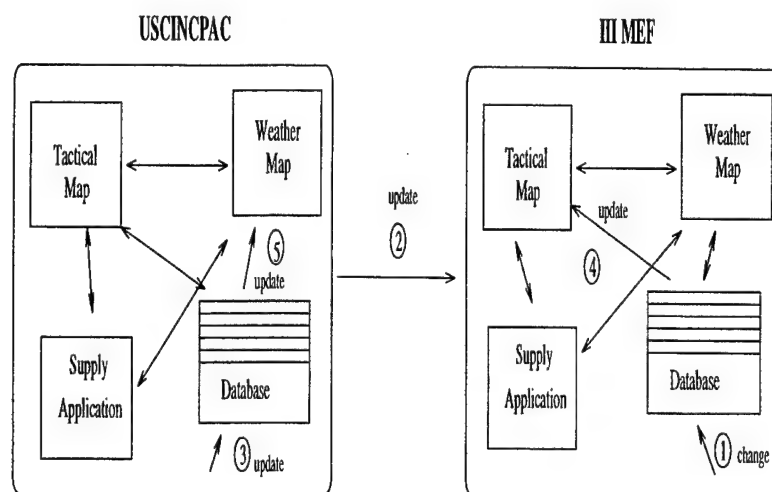
Figure 4. Example of a web represented by the Web Server.

D. INFRASTRUCTURE FOR COLLABORATIVE OBJECT MANAGEMENT, COMMUNICATIONS AND COMPUTING

The bottom layer, Infrastructure: Collaborative Object Management, Communications and Computing, prescribes the required functionality of the hardware and low level components on which the JTF applications operate. Ideally, the JTF Reference Architecture would use a commercially accepted and standardized infrastructure, but currently a mature one does not exist that supports the JTF's needs. However, the fact that the architecture described above is separated into layers allows for the definition of a standard that reflects current industry trends and meets the needs of the

layers above it. If the interface to the bottom layer changes due to a sharp change in current industry practices, layers above it will be partially isolated from it requiring only the interfaces to the infrastructure to be re-implemented.

One function that is needed at the infrastructure level is the management of objects on a distributed network. The reuse of objects and their capabilities will greatly reduce the development time for new applications. Ideally, the users of these objects will be able to use them without worrying about where they are located on the network. Currently, the Object Management Group's Common Object Request Broker Architecture (CORBA) provides this function for the JTF Reference Architecture.

Another major concern about the infrastructure that will support the JTF ATD is the ability of a communication channel to report its specific characteristics. In order for the layers above to properly adapt to changing environments, the bandwidth and latency characteristics must be known to them. The communications environment must therefore be able to supply the needed connection attributes such as a 1-way connection, an asynchronous connection, multiple concurrent connections, or a secure connection. This portion of the infrastructure must also allow for the addition of new services as they become available.

Finally, the computing portion of the infrastructure layer specifies the operating environment in which applications will work. Portability is a key feature needed for this environment. Currently, Microsoft Windows and the X Windows for Unix, the defacto standards, are recommended as the common windowing environments. When incorporating communication between different platforms, the TCP/IP protocol is recommended.

E.   SUMMARY

As mentioned previously, the teleconferencing setup application will be incorporated into the second layer of this JTF Reference Architecture. However, it must use the functionality provided by the layers below in order to be considered compli-

ant. The major concern is making the application adaptable to the changing network environment, and integrating it with the existing functionality of the architecture. In order to accomplish this, we will extensively test the current Communications Server to ensure its proper operation. Early indications show that it is not fully functional. The major problem appears not to lie in the particular implementation of the current Communications Server, but rather appears to be inherent in its software architecture. Consequently, we design a new Communications Server software architecture to facilitate network adaptability. In addition to a Communications Server, we will integrate with an existing application called the TaskTool that directly uses both the Workflow and Socket Trigger Servers.

III. DESIGN OF THE VTC APPLICATION

Currently the users of the JTF ATD software must manually schedule their Video Teleconference (VTC) sessions. The originator of a VTC must contend with many problems such as:

1. Determining a date and time for the VTC that does not conflict with any participant's schedule (usually involving many phone calls).

2. Distributing required materials to all participants. Most materials would be manually encoded on the sender's side and manually decoded on the receiver's side. Actual transmission also requires multiple e-mails from the sender or multiple file transfers on the part of each of the receivers. The JTF ATD architecture has not adopted a standard format such as MIME [Ref. 3] for automatic electronic information exchange.

3. When changes occur in a schedule, more phone calls are necessary.

4. When a document is changed, another round of email messages or more ftp sessions are required.

In addition to reducing the amount of overhead in organizing a VTC, our implementation has the following goals:

1. Take advantage of the existing software services in the JTF ATD Reference Architecture. We can best achieve this goal by integrating our application into the Coordination, Communication, and Control Module of the Application Layer.

2. Our application should be network aware. Being network aware requires, for example, that if a large video will not make it to its destination on time due to a busy network, the VTC application must adjust and send a smaller file, perhaps containing only text, instead.

3. Provide a simple Graphical User Interface (GUI) to select VTC participants (no phone numbers to remember) and to identify pertinent information to be distributed.

We will evaluate the different components of the JTF ATD software to determine both

1. How easy it is to use the JTF ATD infrastructure, and

15

2. whether that infrastructure will enable such an application to meet its adaptivity goals.

We now elaborate on the tools found in the reference implementations of the layers of the JTF ATD Reference Architecture that we will use to build our application.

A. TASKTOOL

In order to meet our GUI goal, we have chosen to use the X11 interface that is embodied in the JTF ATD TaskTool. The TaskTool is an application that is used to send taskers from one person to another. A tasker is a request (or an order) for action. For example, if a commander requires his staff to tell him how many injuries occurred during the last year, he could call on the phone and *task* his staff to obtain this information. Using the TaskTool can reduce the commander's job from multiple phone calls to a few mouse clicks.

Figure 5 shows the initial screen when the TaskTool is run. Located at the top is the login name of the user using this instance of the TaskTool (jpkresho@jtfweb5). The middle portion of the screen shows a list of users that are currently in the user's group. The user can simply choose the person's name to whom he wishes to send a tasker; there are no phone numbers to remember. The bottom portion contains the taskers on which the user must act.

Using our example above, instead of the commander calling his staff on the phone to task them, he will use the TaskTool. In order to initiate a new task, the commander will select the user he wishes to task by clicking on the correct folder (Figure 5). After clicking on the user's folder, another window similar to Figure 6 will appear.

In this window the commander will specify his instructions for the tasker, along with its Due Date, Status, and Priority. Another field that the commander will fill out is the Task Type. The type defaults to Analysis, which signifies that the tasker requests comments on a particular plan of action. Other task types include FYI (For

16
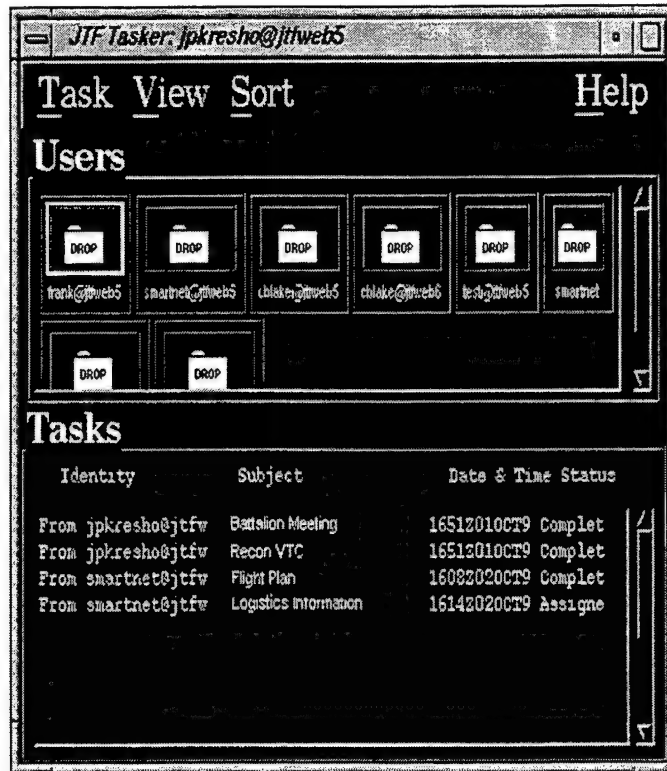
Figure 5. Screen Shot of the TaskTool.

Your Information) and RFI (Request for Instructions). After typing his instructions, the commander can proceed to send the tasker. When the commander hits the send button, the tasker is passed to the JTF ATD WorkFlow Server, which we now describe in detail.

B.   WORKFLOW SERVER

The WorkFlow Server stores and manages objects for the application to which it is bound. For the TaskTool, it handles the flow of all of the data that is input by the users. Once the WorkFlow Server receives this data, it creates an initial tasker object and then tracks the flow of this object between users. In order to design our application, we first had to understand the objects that are created when a tasker is sent.

17

Figure 6. Screen Shot of NewTask for the TaskTool.

1.   JTF ATD Object Definition

To differentiate its objects from objects belonging to other systems, the JTF ATD Reference architecture adopted the Command and Control (C2) Schema [Ref. 4]. The C2 Schema defines a common vocabulary of object classes that all JTF ATD servers and applications share. The key distinguishing factor in the C2 Schema is that it is based on objects, not just data. So, when a specific military object is modeled, the following properties are instantiated:

- Type of the real-world object (e.g., a tasker or vehicle).
- List of characteristics being modeled, such as weight, speed, and color.
- User names for data attributes.
- Semantic restrictions for the data attributes, including constraints and ranges.
- Function methods for the object.

18

- Semantics of the function methods, including pre-, post-, and error conditions.

- Relationship between this object and other objects.

A programmer might model a tank for his software. Using the above list as a guideline, the type of the object would be a *tank*. The programmer would then define several characteristics such as speed, height, length, number of wheels, and color of the tank. Also specified may be that the speed must be measured in knots and the color can only be green or brown. Now that the tank has some attributes and constraints, it can be manipulated with function methods. One such method might be *move*. The programmer can require that before moving, the tank must be stationary, and after the move, the tank is in a different location. Other conditions should also be checked, such as the maximum speed and the quantity of fuel in the tank. Finally, once the tank is modeled, the programmer must also be concerned with relationships to other objects, such as barbed wire. For example, if barbed wire is run over by the tank, the tank will be slowed down until the barbed wire is cleared from its wheels.

2.   Objects of WorkFlow Server

Figure 7 shows the objects (along with inheritance) defined in the WorkFlow Server. As prescribed in the JTF ATD Implementation Guidelines [Ref. 5], each object is defined as a C++ class.

The designers of the WorkFlow Server discovered that all of their objects would contain two similar functions. Therefore, they created the base class C2SchemaObject that contains the functions that place and retrieve data from a common data source (e.g., a database), and determine what kind of object it is (e.g., a Tasker, User, or MultiMedia object). Then the designers name the objects that model the real world objects, such as c2WorkflowUser and c2WorkflowDirectory. We note that the Workflow objects use existing objects as attributes of other objects when appropriate (e.g., A Directory contains many Identities). Applying the guidelines discussed above made writing our application a less daunting task.

3.   TaskTool to WorkFlow

After a user sends a task using the TaskTool, a new instance of the c2WorkflowTasker is created. A method defined for this object called `create` is then executed. Using the information passed from the TaskTool, all of the data members (Figure 7) of the new tasker are initialized. In addition, a c2WorkflowRelation is created, linking the originator and each participant to this new tasker.

At this point, we notice that the C2WorkFlowTasker can be quite useful for our purposes. The data we can store there includes the names of the users to which our application is required to send any attachments, the names of the attachments (and alternative files for use when the C2WorkFlowTasker must adapt to varying network loads), the date of the tasker, and the tasker's priority.

The architects of the JTF Reference Architecture wanted to ensure that, in a distributed system, users on one system are updated when changes occur at another location. For example, when a tasker and a relation are created, the TaskTool needs to update the views of the other users' taskers. Similarly, when another user is added to the system, the TaskTool's view must be updated. The JTF ATD Reference Architecture provides another server for these services: the Trigger Server.

4.   Use of the Trigger Server

The Trigger Server provides the programmer with simple interfaces for the difficult task of managing asynchronous messages. It encapsulates network socket calls, data structures for passing information, and a virtual callback function. The callback function is used when reacting to a view modification. A simple example will illustrate this mechanism.

Figure 8 shows a user viewing a map obtained from the JTF ATD's Map Server. The Map Server associates a trigger with each map object that it distributes. When a map object is changed, a callback function that is specifically written for the Map Server is automatically invoked. The callback function passes an update control message along with the update to the Map Server. Map objects that are in use can be

20

located anywhere on the network because they are CORBA objects: CORBA provides a coherent view of the objects that it manages.

A programmer can create his/her own trigger. For instance, below is the C++ code that defines *my_trigger*:

```
class my_trigger : public virtual X_Stream_Trigger_API
{
public:
my_trigger(XtAppContext &app);
    virtual void callback(CORBA::Object_ptr triggerSource,
char *reason,
c2NameValues_var additionalInfo);
};
```

Passing the Xwindows XtAppContext, which is a pointer to an internal structure used to hold data that is specific to a particular application, as a parameter, ties this trigger to that specific application. The next example defines the corresponding callback function.

```
void my_trigger::callback(CORBA::Object_ptr triggerSource,
  char *reason,
  c2NameValues_var additionalInfo)
{
  cout << reason << " trigger received on "
    << triggerSource->_object_to_string() << endl;
  for (int i = 0; i < additionalInfo->length(); i++)
    cout << "  " << (char *) additionalInfo[i].name << ":   "
      << (char *) additionalInfo[i].value << endl;
}
```

Next, we initialize a structure of the predifined type c2NameValues, which can pass additional information along with the trigger.

Initialization is performed as expressed in the code below:

```
c2NameValues additionalInfo;
additionalInfo.length(2);  //passing two items
additionalInfo[0].name = ``name1´´;
additionalInfo[0].value = ``value1´´;
additionalInfo[1].name = ``name2´´;
additionalInfo[1].value = ``value2´´;
```

We are now ready to "fire" the trigger. We first declare an object of type trigger and associate it with our application, then make a call to the Trigger Server. Here is how such a trigger is declared and associated with an application:

```
my_trigger trigger(app);  // setup trigger object

//now fire the trigger, passing a ``reason´´ and the additional information
Socket_Trigger::_narrow(trigger->info.trigger)->
          TS_Trigger_Fire(trigger->info.trigger, ``reason1´´, additionalInfo);
```

Once the trigger is fired, the callback function will execute and it will output:

```
reason1 trigger received on
:\jtfweb5:Socket_Trigger_Server:4::IR:strigger.idl:Socket_Trigger
  name1:  value1
  name2:  value2
```

In addition to the "reason" and the additional information that we passed to the trigger, there is also some odd looking output beginning with "jtfweb5." This output reveals the identity of the Socket Trigger Object to which the application connected for the service of its trigger. Because the JTF ATD distributed environment permits late binding of trigger to object, CORBA does not return which specific socket trigger object is used until runtime.

The C2SchemaObject class definition already has a virtual trigger defined. Therefore any class that inherits from the C2SchemaObject can define its own functionality. Each object in our VTC application will be derived from the C2SchemaObject class because it descends from the WorkFlow Server. Using careful design and integration with existing WorkFlow objects, we exploit the underlying triggers that are already defined.

C.   BASIC STRUCTURE OF OUR APPLICATION

Our original goal was to design an application that would consist of a server and a client. Our server would track every VTC session that was scheduled. However, this would allow for a single point of failure for the entire VTC scheduling system. Therefore, we decided to have one agent for each user using the TaskTool. Our agent is launched when the TaskTool is initialized, allowing it to update its VTC tracking information using the latest information from the WorkFlow Server.

We decided that we needed a way to track the status of each VTC session, and so we defined a **status file** to contain the following information:

- List of attachments and to which users each has been, thus far, distributed.

- List of participants and whether they have received notification of the scheduled time of the VTC.

- Time at which to clean up files at the participant's location (after giving the participant a chance to save the files).

The name of this status file is the same as the session name, which is discussed later. This status file is stored in the "Planner/VTC" directory of the originator's home directory. The Planner directory is created in the user's home directory by the Planner application when that user is added to the list of the JTF ATD users. The first time a user opens the TaskTool, our agent will create the VTC subdirectory.

Each time the TaskTool is started, our agent must loop through all of the user's taskers and find the VTC taskers. This procedure requires that we first have a pointer to the c2WorkflowUser object that belongs to the specific user. There is a built-in function named *my_identity* that takes as input a login name and returns a pointer to the correct c2WorkFlowUser object (Figure 9). Recalling from Figure 7, the c2WorkflowUser object contains a wealth of information, including copies of all of the taskers associated with the user. Our application will locate most of its required data by starting its search from the c2WorkFlowUser object.

Stepping through the c2WorkflowUser->taskers, our agent looks for any taskers that have the task_type VTC. Once a VTC tasker is found, it is processed with our function `actOnVTC`. The function `actOnVTC` creates a session name for the tasker (Figure 10). The session name is constructed by appending the originator's name to the tasker's due_date. A period is used to separate the name and the due_date, and all blank spaces are replaced by underscores. For example, if the originator is jpkresho and the due_date is 311200Z May 97, then the session name is "jpkresho.311200Z_May_97". Since the originator can only produce one tasker at a time

23

(and all dates use the Zulu timezone), this session name will be unique, and can therefore be used to identify each tasker in the system.

In order to allow our agent software to quickly look up session names, we then insert each session name into a hash table. Using the string_hash object type provided by the JTF ATD architecture makes this a simple job (Figure 10). If the current tasker under consideration is already in the hash table, the agent processes the next VTC tasker.

However, since our agent has not processed this VTC tasker yet, it must determine whether the current user is the originator or the participant. If the user is responsible for the tasker, owner initialization is performed by `initOwnerFile`. This function determines whether the status file already exists from a previous session. If there is no status file for this VTC tasker, it is created and the names of any attachments are written to it (Figure 11).

Once the status file is complete, distribution of the attachments is performed by the function `distribFiles`. For each attachment, a new process is created and run in the background to deliver the attachment(Figure 11). This background process will continue to execute even when a user is logged out.

To ensure that all processes eventually finish properly, even when machines are turned off or a power failure occurs, each computer workstation must keep a log of the currently executing processes. This log will be stored as a file in a system directory that is created when the JTF ATD software is installed. This log tracks the session names which currently have file distribution processes executing. If the computer is turned off, this log file is checked by a daemon upon system startup, restarting any file distribution processes that were interrupted. By using each session's status file, any files which are shown not to be completed will be redistributed.

This method also is used when a file distribution process fails due to a network error or a hardware failure at the distant end. When an error occurs, an error code is written to the status file. This error code will be found within 10 minutes when

24

a timer goes off for the purpose of checking each status file. When the error code is found, a new file distribution process is started (Figure 12).

After the initial pass through the list of taskers for a given user, the agent starts the 10 minute loop for error checking, and then must wait for an event. Currently, the most important event that can occur is that a new VTC tasker arrives before the user logs out. The Trigger Server is an ideal tool to use to react to this event. Since the c2WorkflowUser object contains an array of all of the user's taskers, an addition to this array will cause a change in the c2WorkflowUser object. From the explanation above, we recall that a trigger can be set to react to this change. For example, if User A originates a VTC tasker to User B and User A hits the Send button of the TaskTool, the Workflow Server updates both User A's and User B's tasker array, causing updates of both their respective c2WorkflowUser objects, thus firing a trigger for both users.

We set up the trigger to search for a new task when it is fired. Unfortunately, there is no way to identify the new tasker without iterating through the entire list of taskers for a specific user. A new tasker need not be of VTC type. That is why we used the hash table for storing VTC taskers. When the trigger is fired, the agent loops through the taskers, and when it finds a VTC type, it calls `actOnVTC` which implements the steps described above.

By viewing the contents of Figure 7 and Appendix A, the extensive use of the Workflow objects can be seen throughout all of the functions mentioned above. We also note the extensive use of the trigger and the string_hash object. We therefore conclude that integrating into the existing framework of the JTF ATD Reference Architecture is simple and advantageous for a programmer with several years programming experience in C++.

D.   WISE USE OF BANDWIDTH

Having achieved one of our goals, designing and implementing the agent that can, on demand, distribute the appropriate files for a VTC, we turned to our next

goal: network awareness. To understand network awareness, and the need for it, we consider the following scenario:

A commander wishes to schedule a VTC session with 5 members of his staff 3 hours from now. Three of these staff members are on the same local network as the commander, but the other 2 are located several hundred miles away on a separate network. He uses the TaskTool to schedule the VTC and requires that several attachments be sent to all participants. One of the attachments is a 650MB reconnaissance video. By the time the VTC is about to start, only the 3 local participants have received the video. The wide area link and the distant network are too congested to deliver the rest of the files on time, and there are 2 participants that have no data to refer to.

In order to be adaptive to the network environment, the application should be able to get some form of data to every VTC participant. The unlucky participants in our example should receive at least "still" photos, or in the worst case, a textual explanation of the situation. This will allow them to make informed comments and decisions.

Fortunately, the JTF ATD Reference Architecture provides a Communications Server whose functionality can solve the problem described above. By incorporating into its calculation the size of the file, the bandwidth of the network, and its estimation of the network load, the Communications Server can determine the length of time that it will take to send a given file. Using this server, an application should be able to determine whether a file will fail to arrive on time, and, if so, stop the current file transfer and send a smaller file that will reach the end user in time for the VTC.

Before using the reference implementation of the Communications Server in our application, however, we wrote some small programs to test its use. This was necessary to ensure ourselves that we understood how to correctly use the Communications Server. Chapter IV discusses the Communications Server's design, along with our small test programs. The reason for including the test programs in this thesis is

because they indicated that both the initial design and the current implementation of the Communications Server may not be as useful as it could be in building applications that are truly network aware.
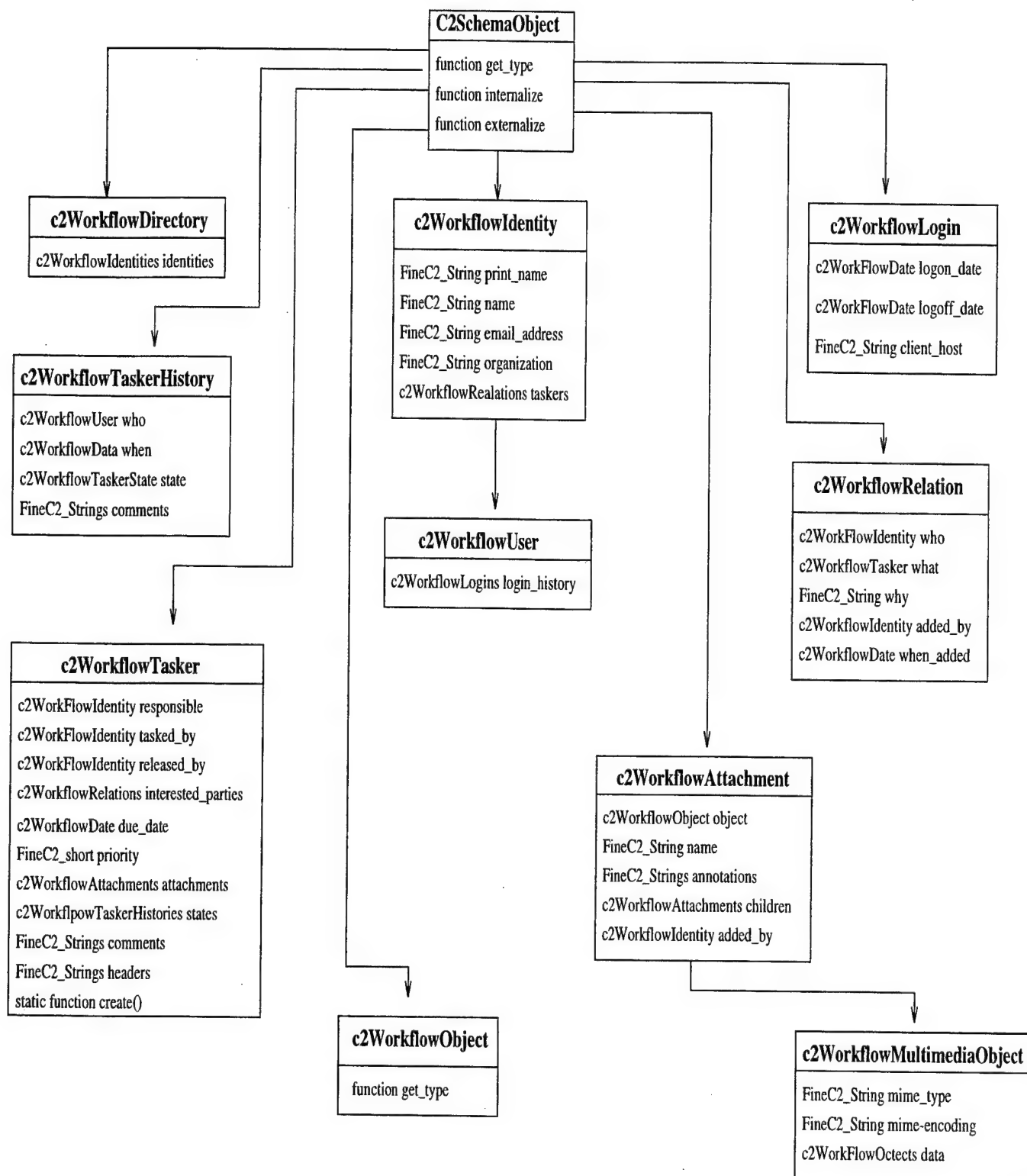
Figure 7. Inheritance diagram of Workflow Server Objects.
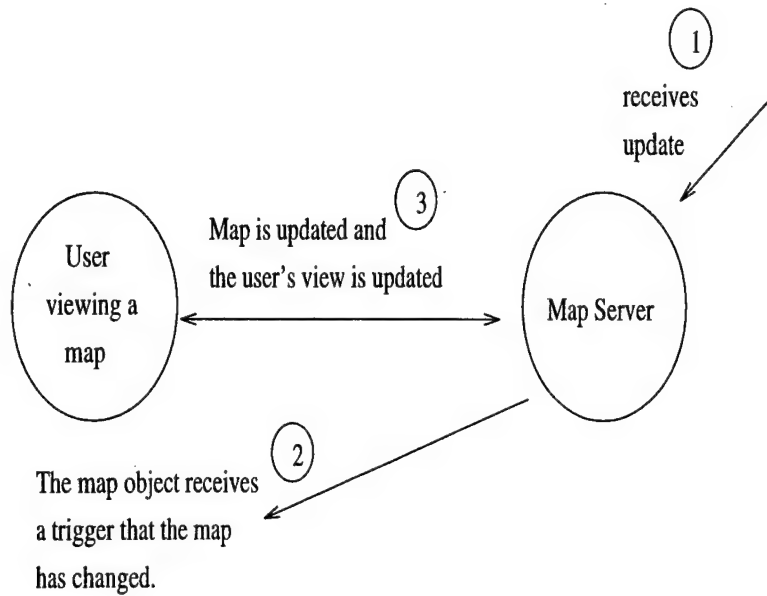
Figure 8. Example of a trigger.

1- A call to my_identity

2 - Block of data for jpkresho obtained from Step 1
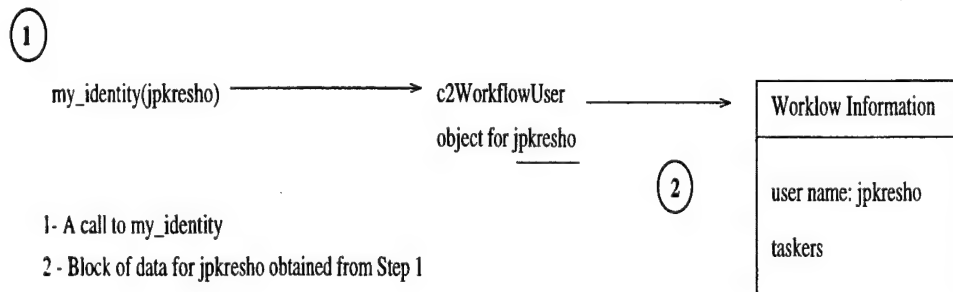
Figure 9. Initial steps when TaskTool is started.

TASKERS

TYPE	ORIGINATOR	TITLE	DUE_DATE
FYI	benton	Intel Brief	241100Z May 97
VTC	jpkresho	Communications	311200Z May 97
FYI	rkarmstr	Artillery Targeting	251340Z May 97
RFI	hensgen	Compiler Problem	220900Z May 97
VTC	jpkresho	Recon Review	260830Z May 97

① ②

session_name = jpkresho.311200Z_May_97 ⟶ ⟦ HASH_TABLE ⟧

③

1 - Loop through Taskers and find VTC taskers
2 - Create session_name
3 - Place into Hash_Table for quick lookup

Figure 10. Finding VTC Taskers and creating session_names.

①

jpkresho = originator ⟶

②

STATUS_FILE

Attachments:
    Recon.movie - NO
    Intel.pic - NO

Distribute
Recon.movie

③

Distribute
Intel.pic

1 - User is originator
2 - Create status file and track distribution status
3 - Create background processes to ditribute attachments
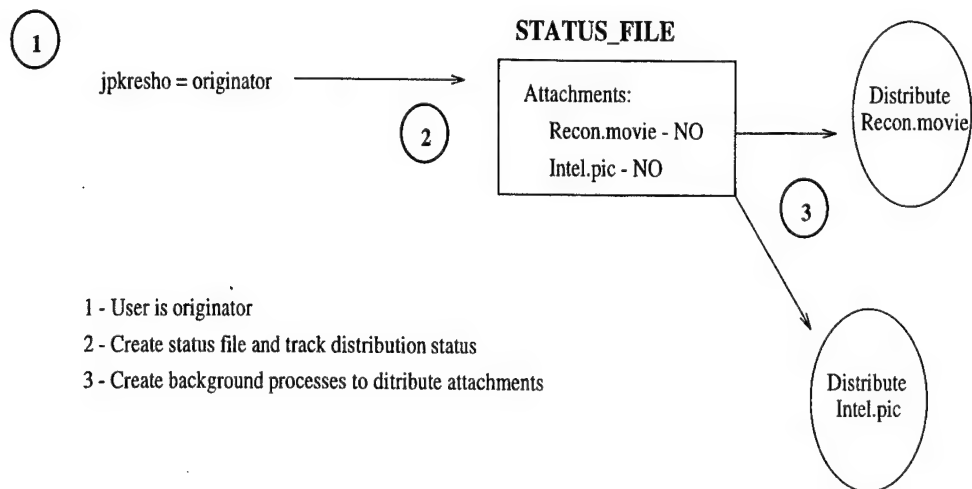
Figure 11. Creation of status file and distribution of attachments.

30
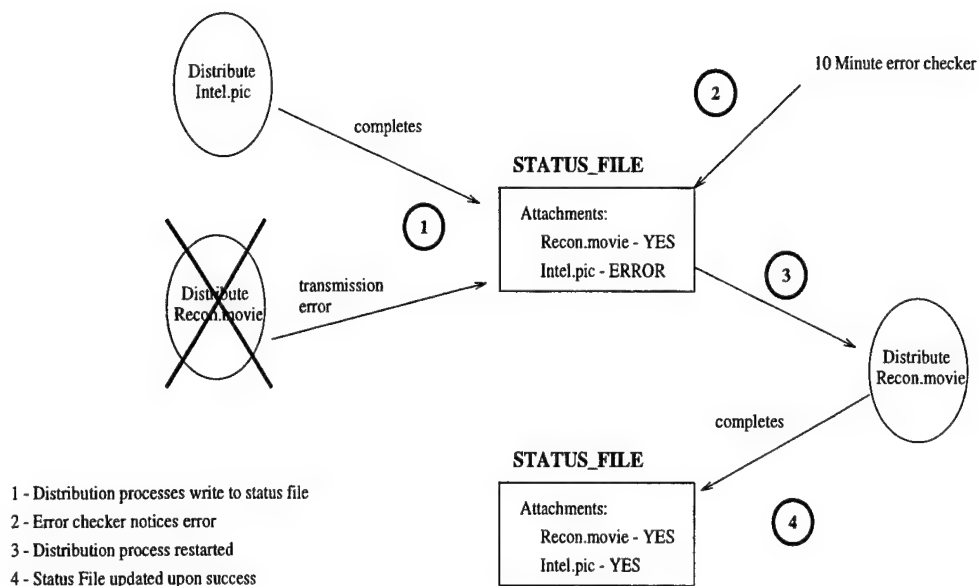
Figure 12. Error detection using the status file.

IV. OVERVIEW AND TESTING OF THE COMMUNICATIONS SERVER

A. OVERALL CONCEPT OF THE COMMUNICATIONS SERVER

The Communications Server brokers limited communication bandwidth to consumer applications via contracts for latency and quality of service (QoS) that are priced to balance supply and demand. In order to be architecturally compliant with the JTF Reference Architecture, any application wishing to provide network adaptivity must use the Communications Server.

There are two goals that the Communications Server strives to achieve for its client applications. The first is to advise applications so that they can easily adapt to the environment that they are currently executing in. The application may need to react differently when it has a bandwidth of 2400 bits/second than when it has a bandwidth of 5 Mbits/second. The Communications Server's second goal is to ensure that higher priority data receives the bandwidth that it needs, while maximizing the probability that the lower priority applications will perform acceptably. One example demonstrating how a lower priority job can adapt to a shrinking bandwidth is for it to send a text file instead of a file with 20 MBytes of graphics and for the receiver to understand which application to execute based upon the data that it receives.

The Communications Server embodies an economic model. When an application wants to send messages over the network, it first queries the Communications Server. For example, initially an application may wish to send 30 MBytes of data and may need it to arrive at its destination within 2 minutes. The Communications Server calculates a cost based on several statistics including the current demand placed on the network resource and the desired latency. When the cost is returned to the application, the application must decide whether it wishes to spend the required amount (does it have enough money?). If not, it may submit a different request, for example,

one with a lower QoS, i.e., a later deadline. The application and Communications Server may iterate these requests several times before determining a QoS that meets the application's budget.

B.   OUR EARLY EXPERIENCE WITH THE COMMUNIC-ATIONS SERVER

The Communications Server must eventually, as described above, be rich in functionality; however, when we started we were not sure precisely what functionality had yet been integrated into the reference implementation. Before integrating our application with the Communications Server, we attempted to experiment with the functions that are defined in its Interface Definition Language (IDL) [Ref. 6].

Before we can describe the functions in detail, we first need to describe a QoS structure that is required by many of them as a parameter. The Communications Server currently defines QoS using a structure called the CS_Flowspec. This structure contains the following information:

- Type of virtual service path required. There are two types currently defined: limited data transfer (e.g., a file) and an indefinite stream of continuous data (e.g., a VTC session).

- The bandwidth of the data flow in bits per second.

- The maximum number of bytes that any individual packet will contain.

- The total size of the data, in Kilobytes, to be transferred. This will be known if a single file (or group of files) is sent. For indefinite service paths, such as VTCs, this field should not be specified.

- The time at which the service path will no longer be needed. This field is used only for indefinite service paths. For single file (or group of files) transfers, this field should not be specified. Time is specified using UNIX's time_t structure.

Another object frequently used by the Communications Server functions is the Service Path object. This object manages an individual data communications flow between a source and destination. The Service Path object contains the following attributes:

34

sp_source and sp_dest. The source and destination endpoints, using a hexadecimal representation of their IP addresses.

sp_fspec. The flow characteristics of the service path, using the **CS_Flowspec** structure above.

sp_negqos. The agreed upon QoS for the service path.

sp_curqos. The current QoS values on the service path.

sp_payerID. The paying client's identification.

sp_totalCost. The total cost for using the service path.

sp_notifyID. The client that should be notified of any QoS change on the service path.

Along with these attributes, the Service Path object provides the following methods that the Communications Server uses:

CS_Spath_StartSpath. Enables the use of the service path. Once the service is started, QoS is measured until the service path is destroyed.

CS_Spath_EndSpath. Terminates and destroys the service path.

CS_Spath_Notify. Notifies the client when a change of service path status occurs.

CS_Spath_UseTokens. Expends currency tokens when data transmissions take place.

The Communications Server's IDL describes several functions that would be very useful to our application. Since our application needed to distribute files and start Video Teleconferencing (VTC) sessions, we chose to first become acquainted with that subset of the functions, which we enumerate below.

CS_CommServer_RequestRev. Allows a user to request a service path between two endpoints. The inputs to this function are the **Flowspec**, an object of type CS_Flowspec, source and destination IP addresses, a start time, and an accounting ID for budgeting. This function returns a **Service Path** object that was set up based on the **Flowspec** and the user's account balance. The **Service Path** will be started based on the input start time,

which could be immediately or days in the future. If there are not sufficient resources to support this request, an exception will be relayed to the client.

CS_CommServer_RequestQoS. Requests an estimate of the current Quality of Service available between two endpoints that are specified by IP addresses. These estimates are based upon anticipated characteristics placed in the **Flowspec**. A Network Quality of Service object is returned, specifying the available bandwidth range, an error rate, and the range of packet delay times currently on the network. If the QoS cannot be determined (i.e., the destination may be down), an exception is raised and relayed to the client.

CS_CommServer_RequestCost. Requests an estimate of a cost of service between two endpoints that are specified by IP addresses, based on anticipated traffic flow characteristics from the **Flowspec** and a requested QoS.

CS_Client_Acct_GetAvailFunds. Determines the current balance of an application's account. All budget information is maintained by a CS_Bank object.

The JTF ATD Reference Architecture requires that the Communications Server be implemented using the CORBA standard [Ref. 7]. In order to become familiar with the Communications Server itself, we first needed to learn how to access CORBA objects. Fortunately, the implementors had some example test code that demonstrated a client binding with the server. The code used to perform the bind to the Communications Server is as follows:

```
CS_CommServer_var cs;

  try {
    cs = CS_CommServer::_bind (":CommServer","");
  }
  catch (CORBA::SystemException& se) {
    cerr <<"Bind to CommServer failed: ";
    cerr <<"unexpected exception" << endl <<se.id() <<endl;
    exit(1);
  }

  cout<<"Got commserver: "<<endl <<cs->_object_to_string() <<endl;
```

From the example code, we see that locating a CORBA object is easily done. We note that we need not specify its machine. CORBA maintains a *repository* that contains the locations of all of the objects that have registered.

However, based on the problems discussed in Appendix B, we have concluded that the CORBA environment for the JTF ATD is very administratively intense, and that this situation needs to be alleviated if the software is to be used in a crisis situation.

1.   Testing the Functionality

In order to obtain a given QoS, we added a call to the CS_CommServer_RequestQoS function to the example code above (Appendix E). First, we initialized the CS_Flowspec parameters required for this function to the following values:

```
flow.CSF_type = CS_sp_singlexfer  // service path = file transfer
flow.CSF_dataRate = 4000000;     // bandwidth of data flow (bps)
flow.CSF_packetLength = 8192;     // Maximum length of packets (bytes)
flow.CSF_totalData = 5000;     // Total data to be transferred (Kb)
flow.CSF_schedEnd = now() + 120   //2 minutes from now
```

We then ran our test program to obtain the QoS readings provided by the Communications Server. Upon completion, the output given to us was:

```
This is the Bandwidth Range:
      Low Val: 797
      Hi Val:  7395
This is the Delay in ms:
      Low Val: 2
      Hi Val:  3
This is the Error Rate:
      Low Val: 0
      Hi Val:  0
This is the Latency in ms:
      Mean:    2
This is the maximum Latency in ms:
      Max:     6
```

The returned values specify the lowest and highest bandwidth that the Communications Server predicts, expressed in Kilobytes/sec. Similarly, the delay shows the lowest and highest values of the expected delay of a single packet in milliseconds. Currently the error rate is 0, but if there were numbers for the Low and Hi values, it would represent the Error rate range expressed as $10^{(-X)}$. For example, if the returned values were:

```
This is the Error Rate:
      Low Val: 3
```

then the Error rate would be 10^{-3} for the Low and 10^{-2} for the Hi. These values indicate that the best an application can expect is that one packet will be lost for every thousand packets, and the worst is 1 packet lost for every 100 packets on average. The final returned values are the negotiated mean and maximum latency requirements for a single packet's transmission time, expressed in milliseconds.

Based on verifying that our input parameters above were not simply echoed by this function, we decided to get an idea of how quickly the Communications Server can react to changing conditions on the network. We wrote a test program that would transfer files between two machines. Initially we assumed that the Communications Server would update itself within 60 seconds. Our original understanding of how the server retrieves its QoS estimates was that it periodically places a load on the network and determines the latency and bandwidth for the data packets that it sends.

Our first test program pair (client and server), Test #1, used the existing ftp protocol. For test program pairs Test #2 through Test #5 we implemented our own file transfer protocol and the code for that implementation is contained in Appendix C. For all tests, the Communications Server was queried by a separate application running on the same ethernet segment as the tests. In each test, client A connected to its host's port 21 (ftp), and then opened up a data connection socket. The corresponding server process waited for the proper "stor" command over the control port. Once this command was received, the server process read a specified file from the file server into a buffer. After this read completed, the entire buffer was then written over the socket. The client process, upon reception of the contents of the buffer, wrote it back to the file server.

Test #1 simply copied 5 files (whose sizes totaled 70 MBytes) from jtfweb3 to jtfweb4, both of which are located on the same ethernet segment. The values returned from the Communications Server for this network segment did not change. We initially thought that the file transfers happened too quickly for the Communications Server to

identify the load. Therefore, in Test #2 we increased our network load by transferring the 5 files repeatedly, 5 times, resulting in a total of 25 files transferred, placing a 350 MBytes load on the network (Figure 13). Again, there was no change in the QoS reported by the Communications Server. Table I shows the size of each of these different files along with actual and predicted times. We timed the file transfers and the average of the actual transfer times is shown in column two. Column three contains the amount of time, based upon the unchanging Communications Server measurements, that the files should have required for transmission.
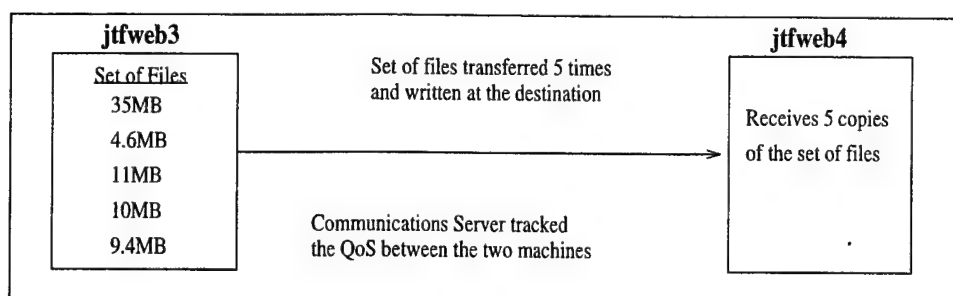
Figure 13. A set of 5 files transferred repetitively 5 times.

FILE SIZE (BYTES)	AVG ACTUAL TIME FOR TRANSFERS(SECS)	PREDICATED TIME FOR TRANSFERS(SECS)
35091968	31	95
4656856	4	21
11039560	10	37
10000640	9	34
9391104	8	33
TOTAL RUN TIME	310 (SECS)	

Table I. Transfer times for different sized files from jtfweb3 to jtfweb4.

We then attempted a third test that involved sending files in both directions between the two machines simultaneously(Figure 14). This test more than doubled the load placed on the network for Test #2. We repeatedly sent the same 5 files as those shown in Table I from jtfweb3 to jtfweb4. Also, from jtfweb4, we repeatedly

39

sent a 35 MBytes file and a 4.6 MBytes file to jtfweb3. The process on jtfweb4 continually transferred files to the jtfweb3 process until the transmission of the 25 files was complete. As a result, the 5 files from jtfweb3 to jtfweb4 had the transfer times shown in Table II.
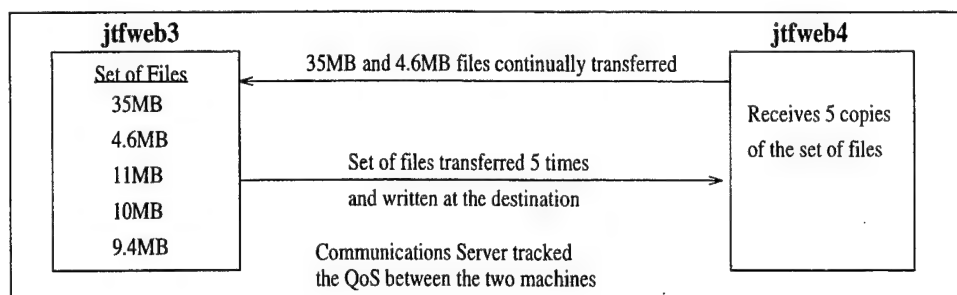
Figure 14. A set of 5 files transferred repetitively 5 times with additional traffic on the network.

FILE SIZE (BYTES)	AVG ACTUAL TIME FOR TRANSFERS(SECS)	PREDICATED TIME FOR TRANSFERS(SECS)
35091968	63	95
4656856	8	21
11039560	23	37
10000640	19	34
9391104	16	33
TOTAL RUN TIME	655 (SECS)	

Table II. Transfer times for different sized files from jtfweb3 to jtfweb4 with additional traffic on the network.

We see that the files in Test #3 suffered a latency double that of the files in Test #2. However, again, there was no change in the QoS reported by the Communication server. The numbers reported were still:

```
This is the Bandwidth Range:
    Low Val: 797
    Hi Val: 7395
This is the Delay in ms:
    Low Val: 2
    Hi Val:  3
This is the Error Rate:
```

```
    Low Val: 0
    Hi Val:  0
This is the Latency in ms:
    Mean:    2
This is the maximum Latency in ms:
    Max:     6
```

We note that the time for the file transfers did not double simply due to increased use of the network bandwidth. Since the test program reads from a file server, into a buffer, then eventually writes back to the file server, several possible resource bottlenecks contributed to the longer latency. The disk access time for retrieving and saving the file must be accounted for. Also, since each machine has only one CPU, only one processing task can be executing at a time on each machine (i.e., reading or writing over a socket). Currently the Communications Server does not account for any activity other than the network traffic. In order to provide an overall picture of the computing environment, another mechanism, perhaps similar to SmartNet [Ref. 8], should be used in conjunction with the Communications Server. Such a mechanism would ensure that all shared resources would be accounted for when determining that a process should adapt.

In order to reduce the delays introduced by disk accesses on the file server, Tests #4 and #5 used an additional test program that reads one large file into a buffer, then transfers it back and forth until interrupted (Appendix D). We used system tools to ensure that we were not swapping back to the file server during these tests. Therefore, after the initial read into a local buffer, there were no accesses performed at the file server. As in the tests above, jtfweb3 and jtfweb4 were used, and a separate application was used to collect statistics on this ethernet segment from the Communications Server.

Test #4 started one server on jtfweb3 that listened for a request for a connection. Then, from jtfweb4, we started the client which connected to the server on jtfweb3 and began transferring a file. The file that we used was 4 MBytes long and took about 8 seconds to make a round trip from jtfweb4 to jtfweb3 and back (Fig-

ure 15). While this test ran there was no other load on the network or on either machine. We started the reporting of the Communications Server's statistics before this transfer began to view the initial load, and it reported the same numbers as above. As we continued to transfer the 4 MBytes file between jtfweb3 and jtfweb4 for a total of 6 minutes, we never saw a change in the statistics reported by the Communications Server.
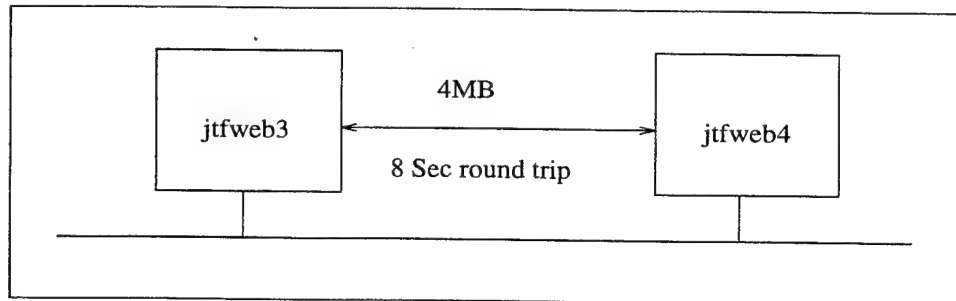
Figure 15. Continuous transfer of 1 file between 2 machines.

In order to produce additional network traffic, Test #5 transferred a 9 MBytes file between vcl and jtfweb5 (Figure 16), two other machines that are on the same ethernet segment as jtfweb3 and jtfweb4. Again, with the Communications Server application tracking the statistics between jtfweb3 and jtfweb4, we began the repeated transfer of the 4 MBytes file. After taking a few readings of 8 seconds for a round trip of this file (same as above), we then started the transfer of the 9 MBytes file. Immediately we could see the time required to transfer the 4 MBytes file almost doubles to 15 seconds. However, the Communications Server never changed the statistics it was reporting. This test also ran for 6 minutes.

At this point we began to think that, perhaps, we were not placing the correct information into the Flowspec parameter that is sent to the Communications Server. Initially we placed the following information into that structure:

```
flow.CSF_type = CS_sp_singlexfer  // type of service path
flow.CSF_dataRate = 2000000;    // bandwidth of data flow (bps)
flow.CSF_packetLength = 8192;     // Maximum length of packets (bytes)
flow.CSF_totalData = 4000;    // Total data to be transferred (Kb)
flow.CSF_schedEnd = now() + 480   //4 minutes from now
```

42
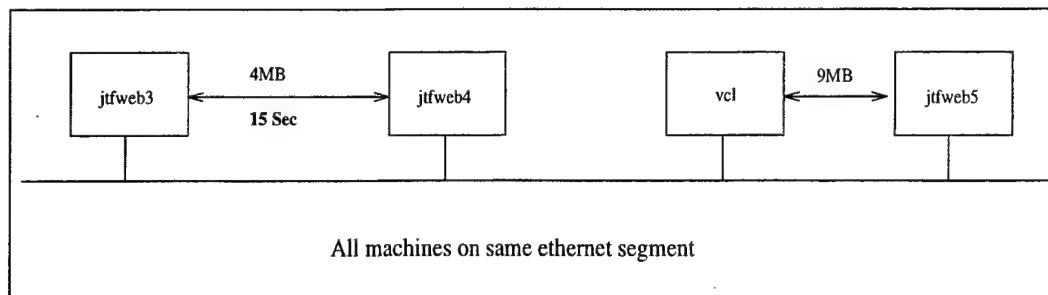
Figure 16. Continuous transfer of 1 file for two sets of machines.

In order to indicate to the Communications Server that we had an even larger load on the network, we changed the bandwidth of data flow to 6 MBytes and the total data to be transferred to 75 MBytes. We then ran the same tests with these new settings, but the Communications Server continued to report the same QoS statistics.

After these tests, we then questioned the Communications Server's implementors again about the results. They thought that we should be getting different values from the Communications Server and they promised to examine the problem more closely. They also stated that the Flowspec structure was currently not being used for QoS calculations, and could be passed to functions without being initialized. This clarification explained why changing these numbers had no effect on the Communications Server. In addition to this information, they also mentioned that the CS_CommServer_RequestQoS function was the only function that was implemented. All other functions currently returned zeros. After further discussion with them, we drew two conclusions about the Communications Server's QoS measurements. First, the QoS was being estimated by having the Communications Server transmit and receive different sized packets and measure the length of time that those packets were taking to travel between two endpoints. We will refer to the placing of data on the network by the Communications Server as a "ping" which is terminology borrowed from a similar UNIX function. Second, the timing of these pings were not frequent enough for our load, which, like all network traffic was bursty, to be reflected in the Communications Server's output.

The implementors re-configured the Communications Server so that it would ping and measure more frequently. Once the frequency of these Communications Server's pings was increased, our test programs found that the Communications Server's output changed. An example of the QoS reports we then received is below:

```
This is the Bandwidth Range:
     Low Val: 2021
     Hi Val:  3385
This is the Delay in ms:
     Low Val: 2
     Hi Val:  20
This is the Error Rate:
     Low Val: 0
     Hi Val:  0
This is the Latency in ms:
     Mean:    2
This is the maximum Latency in ms:
     Max:     6
```

These readings did reflect a heavier load on the network. As one can see, the available bandwidth shrunk and the Maximum Latency went up dramatically. We wanted our application to use the Communications Server to help it decide which files were appropriate to send over the network. However, we did not know how to use the inaccurate raw data reported by the Communications Server. In order to integrate smoothly into the rest of the JTF ATD architecture, we searched for other servers and applications that were using the CS_CommServer_RequestQoS function. We found that several were using a function named cs_qtrans which returned the amount of time it should take to transfer data based on the size of the data. This function simply makes a call to the QoS function, as we did above, and does some calculations based on the bandwidth range.

Unfortunately, using the numbers from the last Communications Server test (Bandwidth low: 2021, Hi: 3385), the cs_qtrans function indicates that a 4 MBytes file would require 57 seconds to be transferred in one direction. Using other numbers from this same test at a different point in time, cs_qtrans returned 360 seconds. However, our tests show that the transfer of this 4 MBytes file, round trip, is only 15

seconds on average, and never requires more than 20 seconds. This difference reveals the instantaneous nature of the statistics gathered by the Communications Server. That is, the Communications Server does not keep any historical data to help identify trends. The Communications Server simply bases its values on a single sample taken, so it might reflect either a sudden brief spike of traffic or a short-lived quiet period.

The Communications Server developers had evidently also found that the current approach to implementing the Communications Server proved inadequate for application developers. That is, their findings must have agreed with ours: the bandwidth and latency predictions varied widely from that which applications could expect to see. Therefore, the Communications Server developers added random number generation to the cs_qtrans function so that application developers could ensure that their applications were adaptive, while awaiting improvements to the Communications Server that could yield better predictions.

C.   CONCLUSIONS FROM THE TESTING

After studying the Communications Server and testing one of its functions, we have made several observations. Currently, the CS_CommServer_RequestQoS function is the only one that returns anything back to the client. The QoS data that is returned from the function is not accurate enough to provide a client with network adaptive capability. In addition, the Flowspec structure is not used, however, it would be a great way to gather additional information about the load on the network. The Communications Server has the potential to be a repository, possibly distributed, of overall network activity.

The method in which the Communications Server gathers its statistics also puts an additional burden on the network. By continually pinging the network, unnecessary traffic is produced. Also, using only instantaneous timings from these pings causes the statistics that it reports to inaccurately reflect the status of the network.

Our original goal was to build a truly network adaptive application. However,

45

the inaccuracies of the information returned by the existing Communications Server prevented us from achieving this goal within the existing JTF Reference Architecture implementation. Therefore we turned our attention toward determining whether adaptation goals could be better met if the Communications Server predictions were more accurate. The following chapters describe additional experiments, and their results, that determine whether a more accurate Communications Server could help applications better adapt at the appropriate time. Following that, we propose an alternate Communications Server architecture that solves several of the problems with the current Communications Server and hopefully can do a better job estimating the network load.

V. SIMULATION EXPERIMENTS

Earlier in this thesis we examined, both qualitatively and through experiment, the ability of applications to adapt, given information from an intrusive server that occasionally examined the state of the resource. That is, the server we investigated placed loads upon resources, calculated the instantaneous performance for those loads, and published this performance prediction to the adaptive clients. Use of this type of server causes two problems. First, it adds to the, already heavy, load on stressed resources. Second, in our experiment with monitoring of network loads, the predicted performance was substantially different from the experienced performance. In this chapter, we investigate, through simulation, how accurate the predictions of resource status, particularly the network resource, must be in order for adaptive clients to obtain good performance. In our simulations, we examine the performance of three different client adaptation strategies, each making use of resource loading information of differing quality. In the next section we enumerate our adaptation strategies. Then we present both our assumptions and the parameters that we varied in our simulations. Finally, we present our simulation results and summarize our conclusions.

A. ADAPTATION STRATEGIES

In different simulations, we varied the percentage of *adaptive clients* between 1.25% and 100% of all clients. Non-adaptive clients exchange data that is available only in a single format. On the other hand, all data that an adaptive client needs to send is available in any of five formats. The actual sizes for each of the five data formats are chosen from exponential distributions with means 3000 MBytes, 300 MBytes, 30 MBytes, 3 MBytes, and .3 MBytes. We assume that our adaptive clients' priorities for the various formats decrease with size. That is, the most important format for each client to send is the largest one and the smallest format is of least importance.

We ran our simulations using three different client adaptation strategies. In

the first strategy, **Strategy 1**, the client first requests a performance prediction from the server; that is, it requests that the server respond with its current estimate of the available bandwidth remaining on the network. The client then calculates whether, based on this predicted bandwidth, it should be able to transmit the largest size format of the required data. If the calculation indicates that this format can be transmitted in its entirety before its deadline, the client begins to send the data. If the client's calculation indicates that this transmission cannot be completed before the deadline, the client iterates through the various size formats from larger to smaller, until it determines the largest one that it can expect to send in its entirety prior to that deadline, and begins to send it. Periodically the server updates the client with new estimates of available bandwidth. If, based upon a new estimate, the client calculates that it cannot complete sending the format that it is currently transmitting prior to its deadline, it stops transmitting that format and searches for a smaller format that it can expect to complete prior to the deadline and begins transmitting that format. [1]

Strategy 2 is very similar to Strategy 1. The only difference is that in Strategy 2, both adaptive and non-adaptive clients take action if their deadline arrives and they have not completed transmitting their current format; they stop transmitting when the deadline arrives. In Strategy 1, such "late" formats were sent to completion despite the deadline having passed.

Strategy 3 acts as a control case. In this strategy, the client does not really adapt. It always attempts to send the largest format and keeps sending it until it has been transmitted in its entirety. We note that this strategy is the default strategy in most Internet web servers.

B.   ASSUMPTIONS

We built discrete event simulations of communication-intensive applications that communicated with one another over a fully-connected network. In this section we

---

[1]The adaptive clients in our current simulation do not ever start sending a larger format.

48

enumerate the values that are fixed in our simulations. Before listing our assumptions about the network, we first define several terms:

Node. A location that contains many computers that generate network traffic, such as a command center or a ship.

Client. An application that generates its own messages. There are typically many clients at a single node.

Best Case Latency. The amount of time it would take a message to arrive if it could use all of the bandwidth on a channel. We will denote the best case latency as L_b.

Given these definitions, our simulation models a network with the following properties.

- There are only two nodes. No routing is done; all messages are sent over direct connections.

- The connection between the two nodes is full duplex with throughput 10 Mbits/second. Half of the network's bandwidth, 5 Mbits/second, is available for each direction.

- Each client using the network receives an equal share of the bandwidth.

During the simulation, non-adaptive clients generate **ordinary messages** according to the interarrival distribution associated with that particular simulation. Ordinary messages differ from **adaptable messages** in that ordinary messages are available in only a single format. The adaptable messages are generated using the same interarrival distribution. All messages have the following attributes.

- A **priority**, P, that ranges between 0 (high priority) and 9 (low priority). We generate the priority using a uniform distribution.

- The **tolerated latency** is the amount of time that the application is willing to wait for the data to arrive, before it considers it to be late. The **deadline** is derived by adding the current time to the tolerated latency. As might be expected, we setup the experiment such that the higher priority messages have smaller tolerated latencies; that is, the higher priority messages need to arrive at their destination sooner. We set the tolerated latency to $a_P * L_b$, where a_P is

49

set using the following criteria. If the priority of a message is 8 or 9, a_P is set to 16; if it is between 5 and 7, a_P is 12; if the priority of a message is between 2 and 4, the a_P is 10; and, finally, if the priority is 0 or 1, a_P is set to 7. The tolerated latencies for adaptive messages are chosen from a uniform discrete distribution of $\{L_b + 30 \text{ minutes}, L_b + 60 \text{ minutes}, L_b + 90 \text{ minutes}\}$.

- Non-adaptive clients send ordinary messages that have different lengths depending upon their class. Class A messages are exponentially distributed around 1 MByte; Class B messages are exponentially distributed around 1.4 MBytes; and Class C messages are exponentially distributed around 50 MBytes. In addition, classes of messages are generated with different frequencies. Class A messages are generated 60% of the time, Class B messages 25% of the time, and Class C messages 15% of the time.

The above assumptions are similar to those used in simulations of the Communications Server performed by Tecknowledge Federal Systems [Ref. 9].

C.   SIMULATION PARAMETERS

In this section we identify the various simulation parameters and how we varied them over different simulations.

We ran different simulation experiments for different average interarrival rates. In each simulation, the amount of time between node message generation is exponentially distributed around the mean. The means for the different experiments were set at 2 seconds, 3 seconds, 15 seconds, and 60 seconds.

In each experiment, some of the clients were adaptive, while the remainder were non-adaptive. We ran different experiments where 100%, 20%, 10%, 5%, 2.5%, and 1.25% of the clients were adaptive.

We also ran different experiments varying the accuracy with which our server could predict the instantaneously available bandwidth. Since our simulated server knew the exact instantaneous bandwidth available, to vary the accuracy, we chose a number from an exponential distribution around various different means and with probability .5 we added the generated number to the actual instantaneous bandwidth, otherwise we subtracted the generated number. We ran different experiments for these

50

means, measured in Kbits/second, of 0, 2.5, 5, 7.5, 10, 20, and 50. We call this the *Instantaneous prediction.* The majority of our experimental space (Figure 17) was focused on the three parameters discussed above.
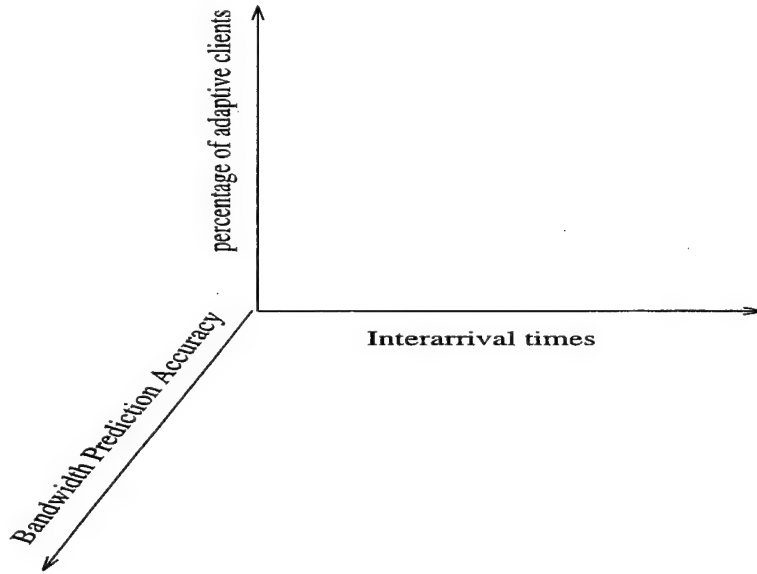
Figure 17. Experimental space using the parameters of interarrival times, percentage of adaptive clients, and accuracy of bandwidth estimates.

Also, we had our simulated server use two different sets of **weights** when producing its bandwidth estimation. In the first case it used $Prediction_i = 0.15 * Instantaneous\ prediction + 0.85 * Prediction_{i-1}$, where $Prediction_i$ was the estimate used to determine if the current format could be completed. We also performed experiments using $Prediction_i = 0.85 * Instantaneous\ prediction + 0.15 * Prediction_{i-1}$. When we discuss results pertaining to using these different weights in Section 2, we will denote them as the weights (.15, .85) and (.85, .15), respectively.

In addition to the above predictions of bandwidth, we also ran experiments simulating the bandwidth predictions of the JTF ATD Communications Server. We now describe how we arrived at the values we used in those simulation experiments.

1. Communications Server Bandwidth Prediction Values

In order to use accurate information for our simulations of the JTF ATD Communications Server, we ran several file transfer experiments. While the Communications Server monitored the bandwidth between two computers, we transferred files and retrieved 35 readings of the Communications Server's predictions. With these data points, we fitted the curve representing the difference between actual bandwidth and predicted bandwidth, to an exponential distribution with a mean of 1Mbit, and offset 340Kbits in the negative direction. This distribution was used to produce bandwidth predictions for simulating the Communications Server.

2. Random Seeds Used

We ran each experiments described above for 10 different sets of random seeds. In each set, a different seed was used for each of the following distributions:

- the interarrival rate,
- the message class type generation,
- the distribution that determined whether a client was adaptive or non-adaptive,
- each different message size distribution, and
- the priority.

D. RESULTS

In this section we present results from our simulations and summarize our conclusions from these results. We present results from additional simulations in Appendix F. The data in this appendix is consistent with the conclusions that we draw in this section. For each experiment, we measured both the average size of the adaptive messages that arrived before their deadline and the percentage of adaptive messages for which no format arrived on time.

1. The Need for Adaptation

We first present results that demonstrate the need for adaptation. After these results and our conclusions from them, we restrict our attention to only Strategies 1 and 2 (Section A).

As mentioned above, Strategy 3 was our control case and did not use adaptive applications. In these experiments, 5% of the processes attempted to send a message in size similar to the first form of an adaptive message, 3000 MBytes. For all interarrival times (means of 2 seconds, 3 seconds, 15 seconds, and 60 seconds), 98% of these large messages did not arrive before their deadline, even when we removed messages immediately if they exceeded their deadlines. In order for these applications to meet critical deadlines,, it is apparent that both a method of estimating the network resource load and a strategy for adapting to bandwidth availability is needed.

2. The Effect of Varying Weights

In these experiments, we found that using the weight pair (.15, .85) is substantially better than the pair (.85, .15). In later sections we restrict our discussion to experiments involving only the weight pair (.15, .85).

Using Strategy 1, we ran simulations where 5% of the messages were adaptive to compare the different weighting schemes. Table III shows these results. The

INTERARRIVAL (SECS)	SIZE FOR (.85, .15) (KBYTES)	(.15, .85) (KBYTES)
2	761	1271
3	1833	2546
15	45793	45929
60	297269	299379

Table III. Average size of messages received using different weighting schemes under Strategy 1 (5% applications adaptive).

weighting scheme (.15, .85) is much better when the messages have an interarrival mean time of 2 and 3 seconds, and slightly better than (.85, .15) for the 15 and 60

second mean interarrival times. The difference between the weighting schemes is a matter of reaction time. Using (.85, .15), adaptive applications will tend to react more quickly to an instantaneous reading which can cause resource thrashing, especially in a heavily loaded environment. On the other hand, the (.15, .85) scheme allows adaptive applications to make better informed decisions based on statistics gathered over a period of time. Based on these results, we ran the rest of our simulations using the (.15, .85) weighting scheme.

3.   Strategy 1 vs. Strategy 2

In this section, we see that there is some benefit to be gained from dropping messages that exceed their deadline. In comparing these two strategies, we use an *Instantaneous prediction* with the mean difference between actual and predicted bandwidth being 5.0 Kbits.

Figure 18 shows that when the network resource is very busy (mean interarrival rate of 2 seconds), the benefit of dropping messages that exceed their deadline are substantial. The greatest benefits are seen as more of the applications using the network resource cannot adapt. The results for a 3 second interarrival time are very similar.

When the network resource becomes less loaded, there is less benefit from dropping late messages. This is due to the fact that there are fewer messages that are late, and hence eligible to be dropped, because the network resource is not in high demand. Figure 19 shows the result for mean interarrival time of 15 seconds. When the mean interarrival time is 60 seconds, applications receive sufficient bandwidth the majority of the time. Figure 20 demonstrates that in this case there is no benefit or penalty from dropping late messages.

4.   Determining How Accurate Server Estimates Should Be

In this section we examine a multitude of operating points to determine under what conditions:
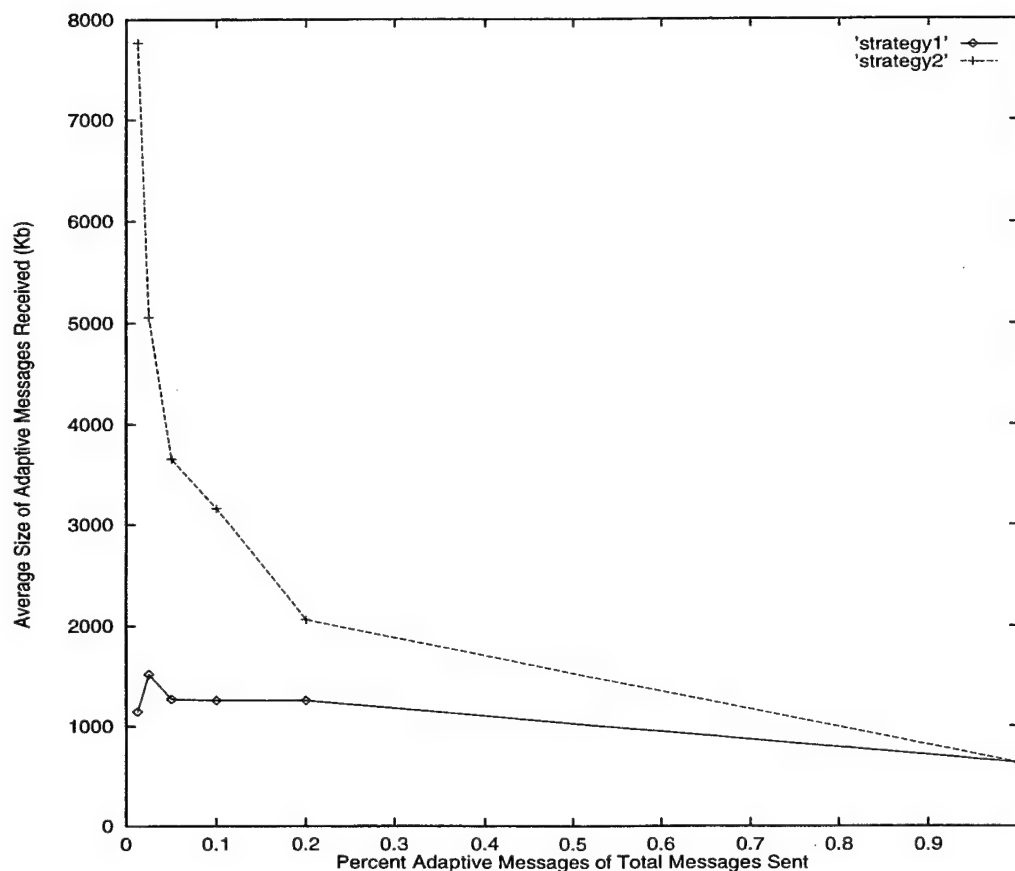
Figure 18. Average size of adaptive messages received for an interarrival time of 2 seconds and mean delta bandwidth prediction of 5.0 Kbits.

1. A simple server, such as the JTF-ATD Communications Server, will suffice, and

2. When a more accurate assessment of resource load is needed.

Before discussing actual results for different accuracies of server estimates, we note that simulations that use 2 and 3 seconds as their mean message interarrival time model a crisis situation. In a military environment, such interarrival rates occur in an emergency, such as during a sudden biological attack. In this case, the network resource will be in high demand, but the priority messages must make it to their destinations before their deadlines. However, when the mean interarrival times are 15 and 60 seconds, the network resource is under normal use, and not many applications are competing for the same network resource.

55
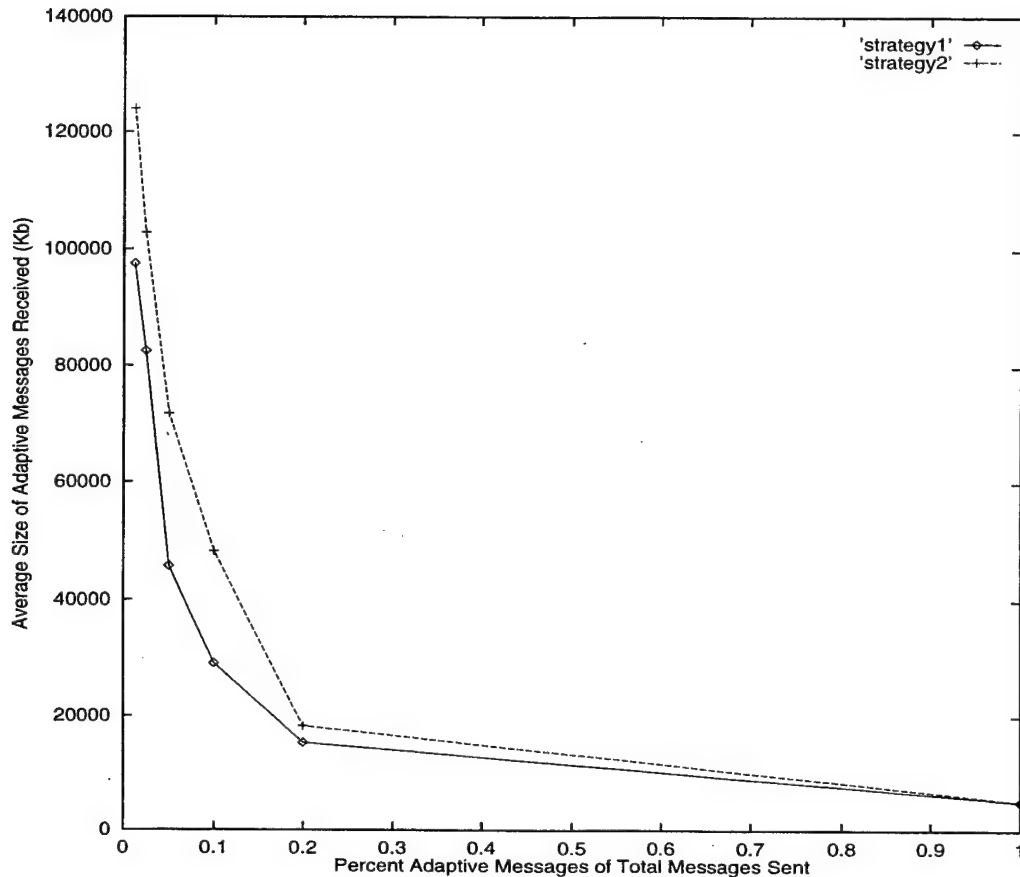
Figure 19. Average size of adaptive messages received for a mean interarrival time of 15 seconds and mean delta bandwidth prediction of 5.0 Kbits.

In order to determine how accurate a server must be when estimating a resource's load, we collected data for each of the *Instantaneous prediction* means enumerated above. The resulting difference in bandwidth accuracy is defined as:

$$Delta\ Bandwidth\ Prediction\ =\ |\ predicted\ bandwidth-\ actual\ bandwidth|$$

The first criteria analyzed was the number of messages that did not make their deadlines under Strategies 1 and 2. Figures 21 and 22 display the results of Strategy 2 when there are 100% and 1.25% adaptive messages respectively. Note that the points labeled "Communications Server" model the accuracy of the JTF ATD Communications Server as discussed above. The results showing the number of late messages for Strategy 2 show a similar trend to Strategy 1. These results are shown in Appendix F.

56
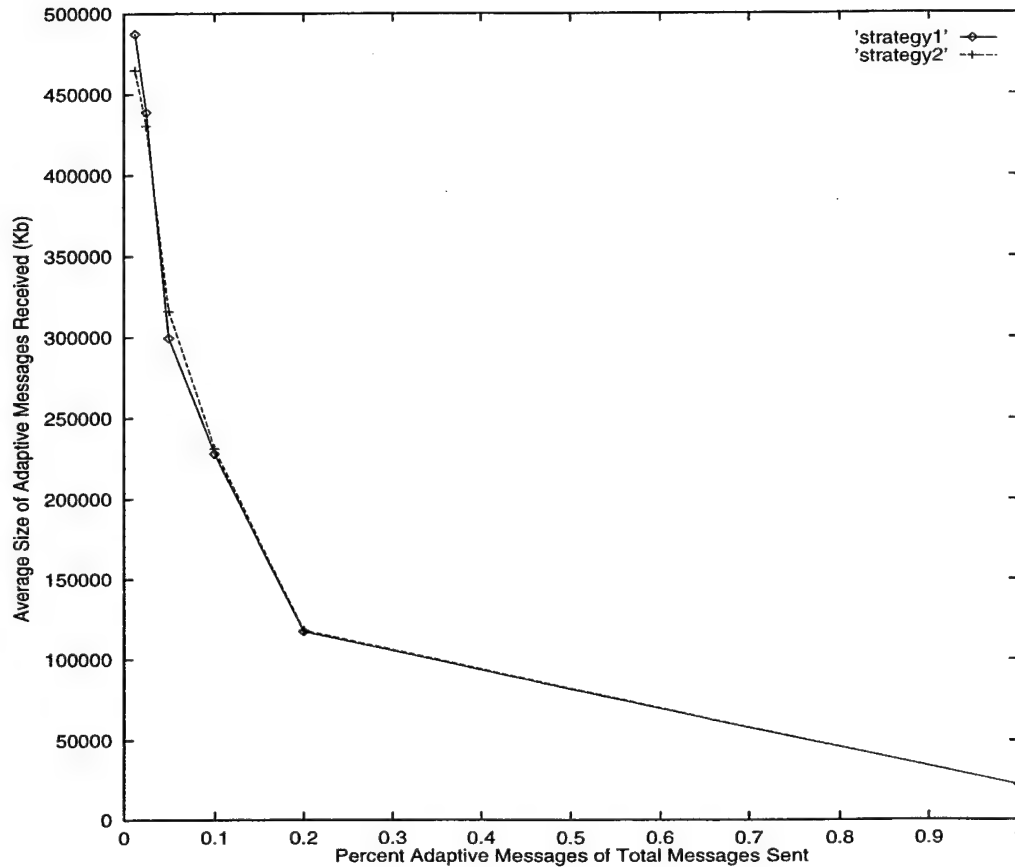
Figure 20. Average size of adaptive messages received for an interarrival time of 60 seconds and mean delta bandwidth prediction of 5.0 Kbits.

For each of the *Delta Bandwidth Prediction* means, including the Communications Server, there were no late messages for mean interarrival times of 15 and 60 seconds. Therefore. we now focus on the crisis situations (2 and 3 second mean interarrival times) to determine how accurate a server's prediction must be. Figure 23 and Figure 24 eliminate the Communications Server reading and focus on more accurate assessments. Again, the results for Strategy 1 show similar trends and can be viewed in Appendix F.

After examining these results closely, we determine that in a crisis situation, being within 5 Kbits/second of the actual network throughput allow most messages to meet their deadlines. Using a less accurate server results in a significant amount
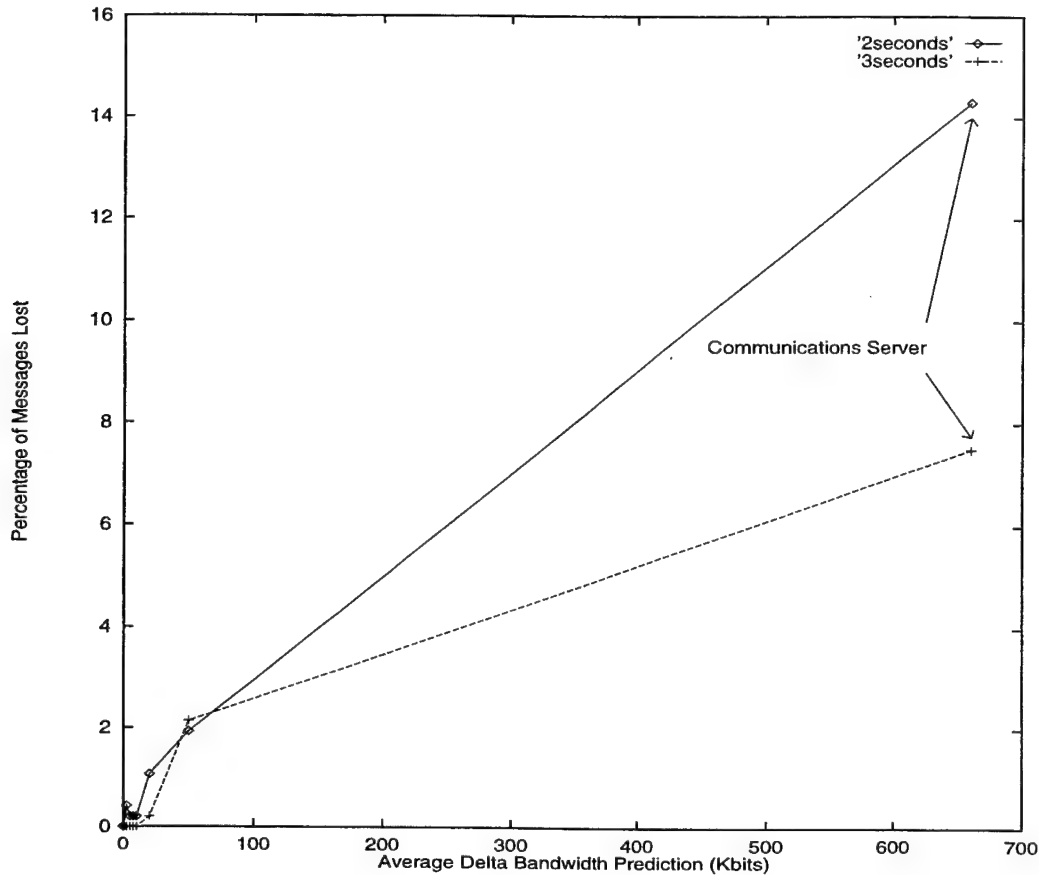
Figure 21. Percentage of adaptive messages not received by deadline when using Strategy 2 and 100% of messages are adaptive.

of lost data.

The second criteria we examine is the average size of the message that does arrive on time. Figure 25 shows the results for Strategy 2 when 100% of the applications are adaptive. We note that when the server estimates are less accurate, only smaller messages are successfully received. This trend follows for all interarrival times for these simulations, and those results can be viewed in Appendix F.

In order to better understand the circumstances under which the average size received is maximized, we refer to Figure 26. The figure indicates that being within 2.5 Kbits/second will get the best results in a crisis situation under most loads. The only exception to this rule is seen when only 1.25% of the applications are adaptive,
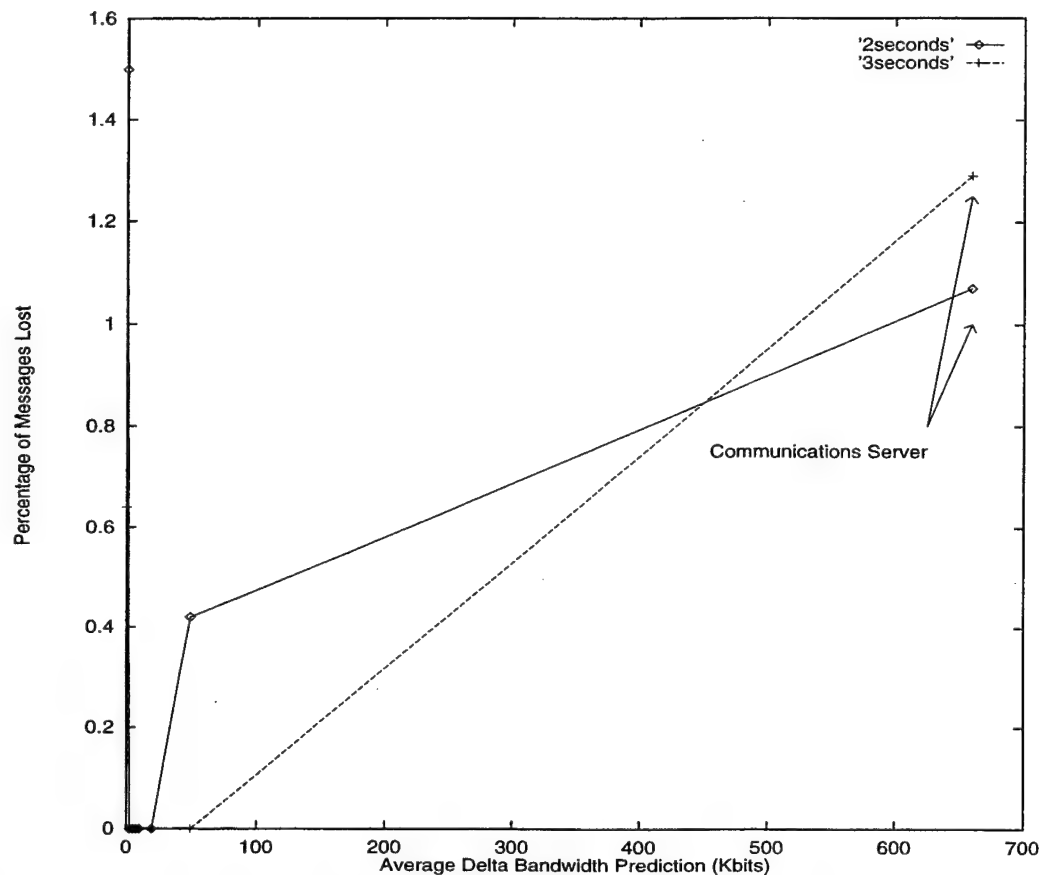
Figure 22. Percentage of adaptive messages not received by deadline using Strategy 2 and 1.25% of messages are adaptive.

and the mean interarrival rate is 60 seconds. Figure 27 shows that in this case a server such as the Communications Server will allow for larger adaptive messages to arrive on time. Since there is little competition for the network resource, a less accurate picture of the load is acceptable. Overall though, as Figure 28 shows, when a crisis situation occurs, it is better to have an accurate server, one that can predict the network bandwidth within 5 Kbits/second. The results for Strategy 1 again are similar to the results presented here. Appendix F contains all the results for these simulations.

As can be seen from the results, most cases require an accurate estimate of the network resource (within 5 Kbits/second), especially in a crisis mode. However, when
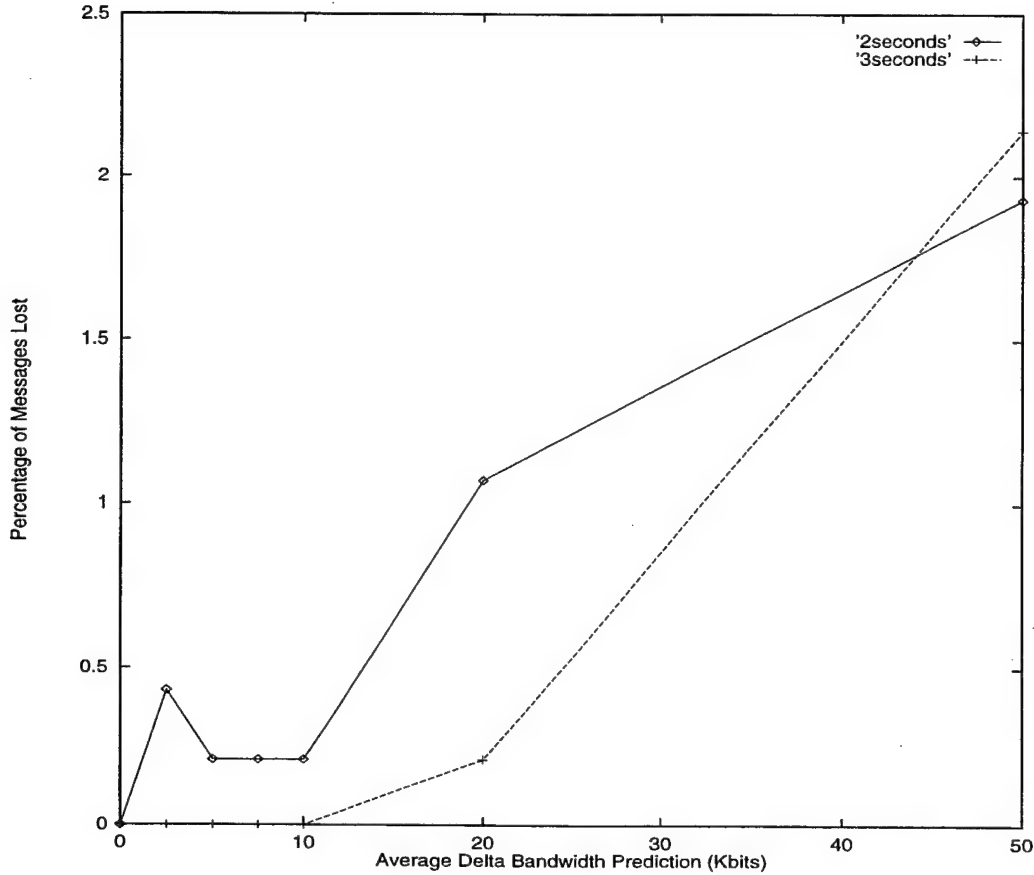
Figure 23. Percentage of adaptive messages not received by deadline using Strategy 2 and 100% of messages are adaptive.

there is little competition for the resource and the percentage of adaptive applications were small, a less accurate estimate may be useful.

E.   CONCLUSIONS

In this chapter we saw that for many situations, an adaptive client requires a better resource load assessment than can be furnished by an intrusive server that occasionally examines the state of resources. In Chapter VII we describe a proposed architecture for a new server that we expect to yield this better assessment. However, before we propose an architecture, we formalize the scheduling problem to which that architecture must contain a solution. In particular, although scheduling algorithms
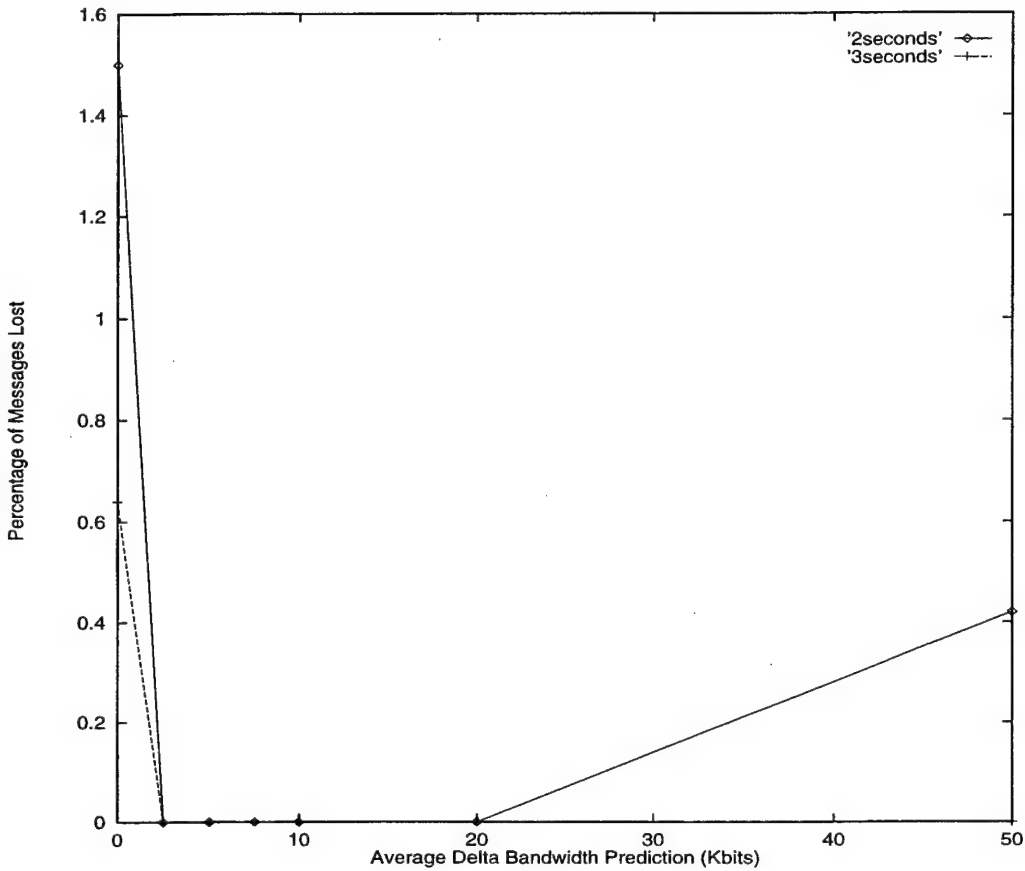
Figure 24. Percentage of adaptive messages not received by deadline using Strategy 2 and 1.25% of messages are adaptive.

are beyond the scope of this thesis, our architecture must ensure that the scheduling algorithms that the architecture will contain are furnished with the information they need by the rest of the architecture. To determine the information that must be provided by the architecture to these algorithms, we next formalize the scheduling problem.
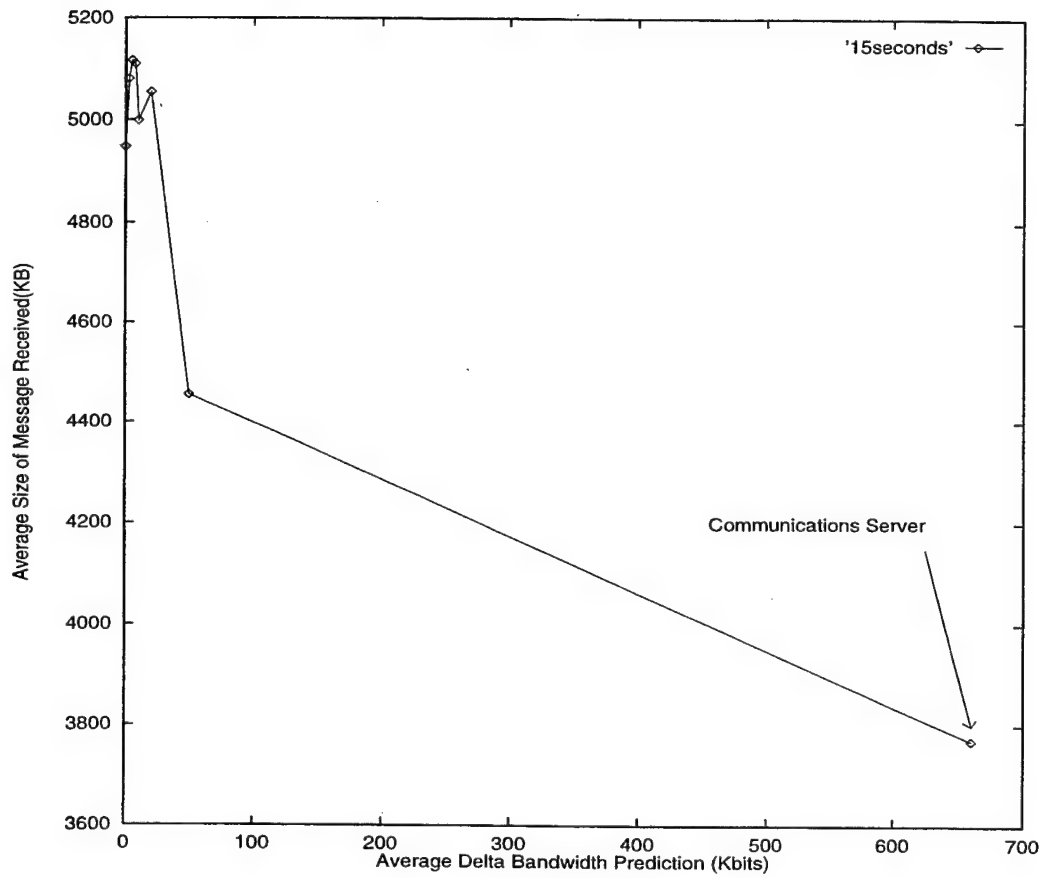
Figure 25. Average size of successful adaptive messages using Strategy 2 when 100% of the messages are adaptive and the mean interarrival time is 15 seconds.
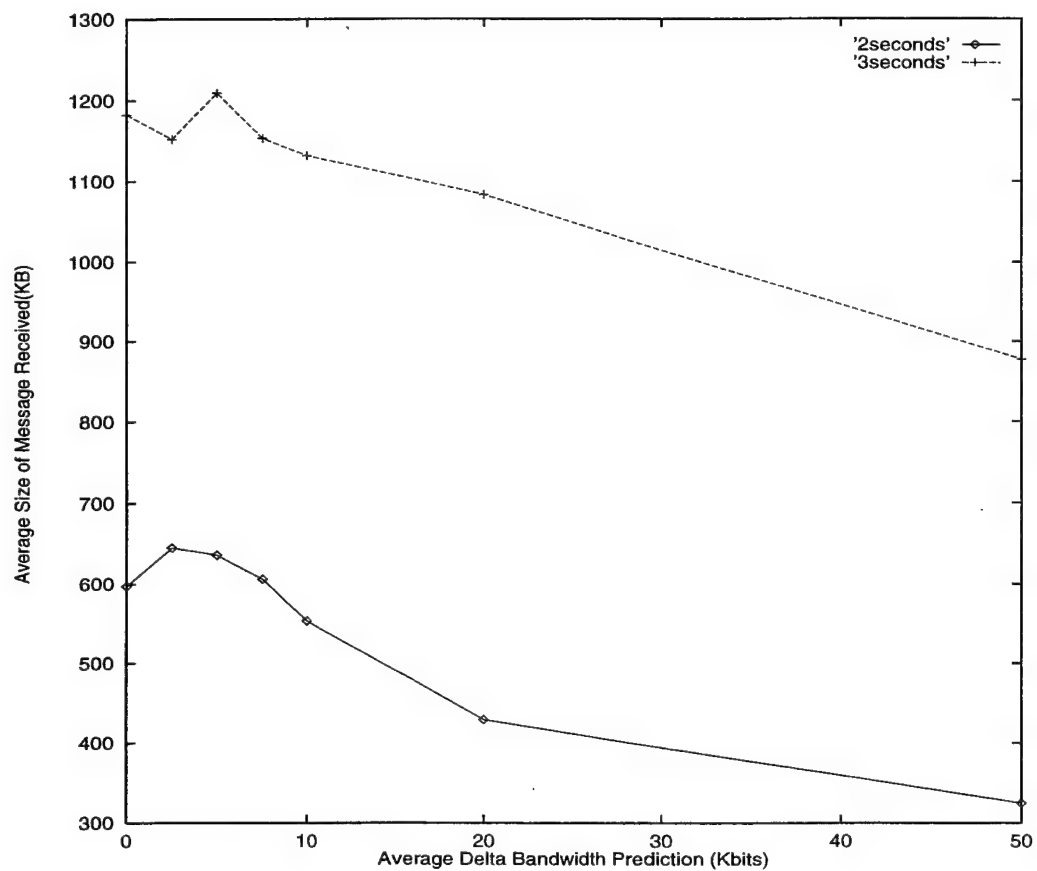
Figure 26. Average size of successful adaptive messages using Strategy 2 when 100% of the messages are adaptive and the mean interarrival times are 2 and 3 seconds.
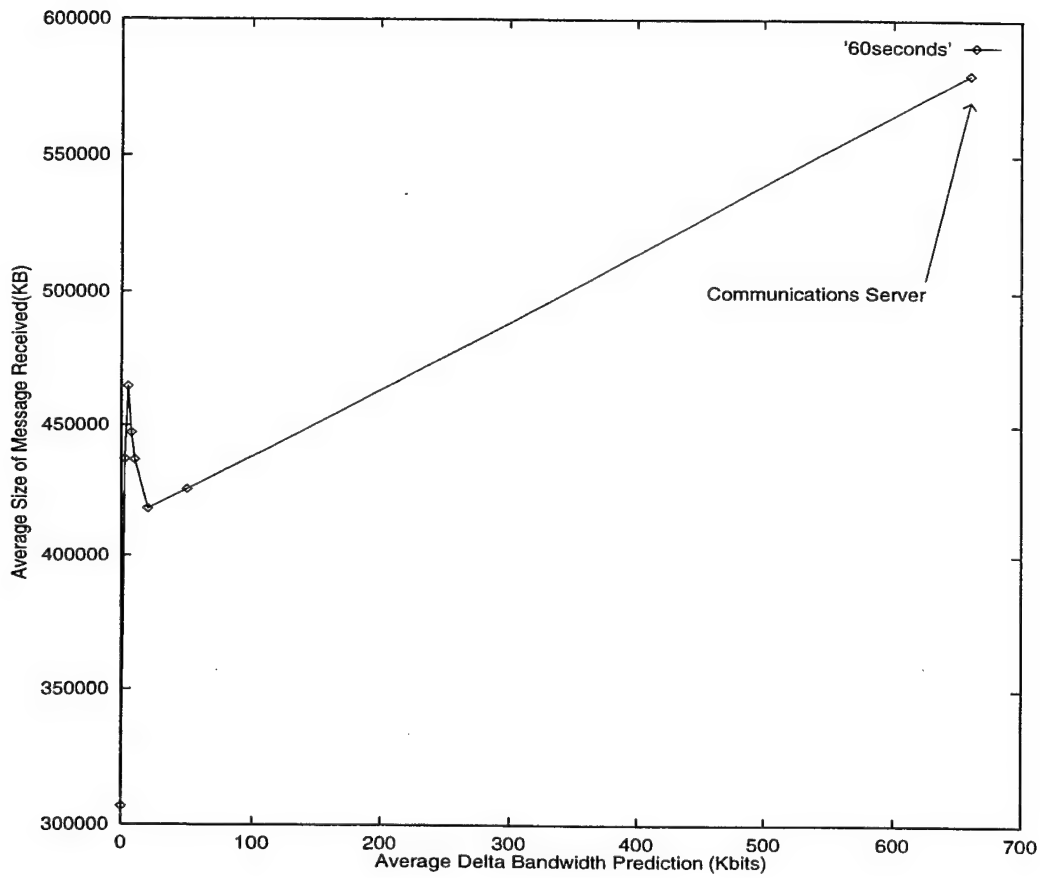
Figure 27. Average size of successful adaptive messages using Strategy 2 when 1.25% of the messages are adaptive and the mean interarrival time is 60 seconds.
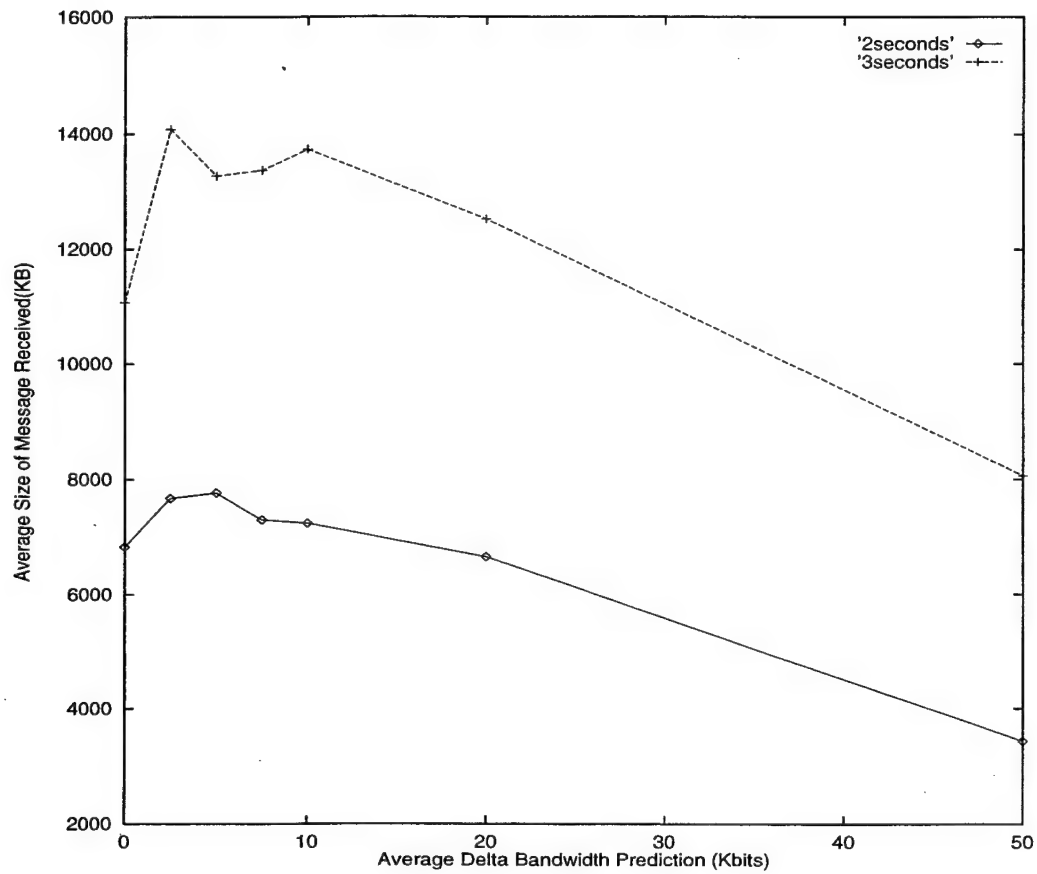
Figure 28. Average size of successful adaptive messages using Strategy 2 when 1.25% of the messages are adaptive and the mean interarrival times are 2 and 3 seconds.

VI. MATHEMATICAL FORMULATION OF THE PROBLEM

In this chapter, we formalize the mathematical problem that must be solved in order that prioritized adaptive applications will receive the best quality of service, given the total demand for resources. The previously described simulations examined the question of how well we must know the load on a (network) resource in order to build adaptive applications that use that resource. Before presenting our mathematical model, we demonstrate that many resources, not just the network, must be considered in such a model. In fact, we show that other resources must be considered even if the applications are exclusively network-intensive. Therefore, we first demonstrate the need to monitor these other resources, such as CPUs and hard drives, then state the formal definition of the problem any architecture that supports adaptive applications must address.

A. ADAPTIVE APPLICATIONS NEED TO KNOW ABOUT ALL RESOURCES

In addition to the load on the network, other resource loads must be monitored to give an application an effective picture of its current computing environment. If a network intensive application ignores the use of resources such as CPUs and hard drives, it will have an inaccurate estimate of when a particular file transfer will end, causing data to sometimes miss a deadline and therefore to be useless at the receiver's end.

In order to show that the load on both the CPU and the hard drive affect a network transfer, we conducted experiments that transferred files between two computers connected via an isolated ethernet network (Figure 29). The two computers were identical Intel Pentium Pro computers running Linux, a PC version of Unix.

Three different sized files were transferred from Computer A to Computer B
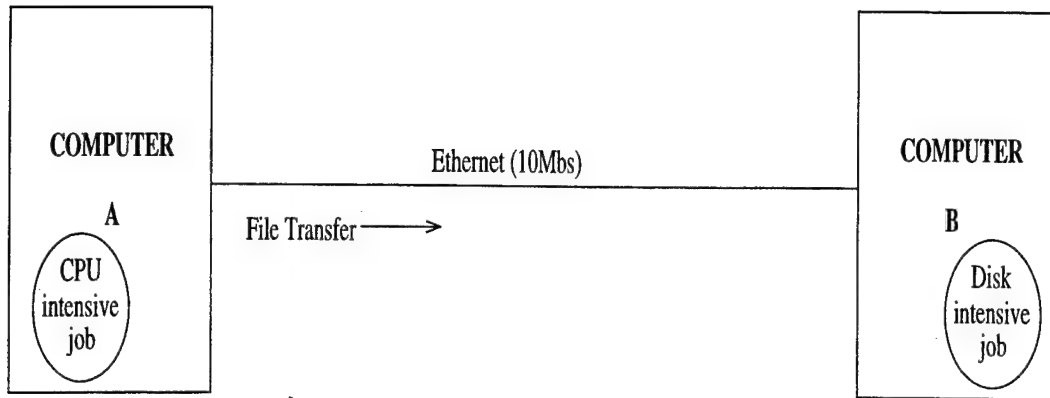
Figure 29. Measuring transfer times with different loads on the computers.

(5MB, 15MB, and 50MB). For all experiments performed, these file transfers were the *only* loads placed on the network.

In order to simulate CPU and disk intensive processes, we placed BYTE Magazine's BYTEmark benchmarks [Ref. 10] on each computer. Initially, a file of each size was transferred from Computer A to Computer B. Neither machine was executing any programs besides the the ones listed below. The following list of experiments were performed while transferring a file of each of the sizes enumerated above:

1. No programs other than the file transfer were executing on either Computer A or Computer B.

2. 1 CPU intensive process on Computer A

3. 1 disk intensive process on Computer A

4. 1 CPU intensive process and 1 disk intensive process on Computer A

5. 1 CPU intensive process and 2 disk intensive processes on Computer A

6. 2 CPU intensive processes and 2 disk intensive processes on Computer A

Each of the above experiments were then also performed with the loads in experiments 2-6 on Computer B, and no load on Computer A. Individual experiments were run 10 times to gather the data discussed below.

68

Table IV shows the results of these experiments when the loads were place on Computer A, and Table V shows the results when the loads were on computer B.

Size (MB)	1CPU (%)	1DISK (%)	1CPU,1DISK (%)	1CPU,2DISK (%)	2CPU,2DISK (%)
5	1.35	8.93	10.67	25.37	15.52
15	0.02	12.00	11.49	26.14	25.75
50	0.13	8.73	9.67	12.60	21.03

Table IV. Percentage increase of time required for file service, under different load conditions for sender.

Size(MB) (MB)	1CPU (%)	1DISK (%)	1CPU,1DISK (%)	1CPU,2DISK (%)	2CPU,2DISK (%)
5	1.23	11.7	13.57	20.77	21.10
15	1.61	8.54	12.65	25.72	16.87
50	1.45	7.22	12.35	14.26	16.90

Table V. Percentage increase of time required for file service, under different load conditions for receiver.

We then simultaneously placed different loads on Computer A and Computer B at the same time while transferring each file. Table VI shows the results from these experiments.

Size(MB) (MB)	1CPU,1DISK ON A AND B A AND B(%)	2CPU,2DISK ON A AND B A AND B(%)
5	20.73	37.70
15	23.22	36.11
50	17.8	30.05

Table VI. Percentage increase of time required for file service, under different load conditions for both sender and receiver.

Because the file transfer is the only traffic on the network, the longer transfer times shown in the above tables is due to the use of other resources. Even though

69

the file transfer is network-intensive, it must obtain the CPU resource to execute instructions which move the data to memory and over the network. Therefore, it competes with the compute-intensive process for that resource, effectively slowing *both* processes. These experiments show that not only does the network resource need to be monitored, but also other resources such as CPUs, memory and hard drives must be monitored to predict performance of even network-intensive processes. If the transfer of an important video file is to take place under loads similar to that shown in Table VI, then an estimate of three hours based only on an accurate assessment of the network traffic would underestimate the time required by an hour. Because we expect to support both network-intensive and other applications, e.g., real-time, compute-intensive, and I/O-intensive applications, our formal description of the problem at hand includes all relevant resources and not simply the network.

B. THE FORMAL MODEL

In the next chapter we will propose an architecture for supporting adaptive clients that incorporates mechanisms for obtaining feedback on the status of resources as well as providing scheduling advice for the processes that must be adaptive. In our model, we assume that we have various applications running on a set of distributed, shared and heterogeneous resources. The problem that our new architecture addresses is that of how to provide a software architecture that permits these applications to *best* adapt to varying loads on those resources.

Our mathematical model formalizes what we mean by *best*. Each application uses resources to acquire needed information such as weather data, map data, and/or planning data. The process of acquiring this information can be very simple, requiring only a query of a database, or it can be quite complex, requiring that many different pieces of data be gathered together, used in a simulation, and the results of that simulation be analyzed. In each case, the needed information may be available in many different forms, e.g., fully rendered videos, color graphics, and textual summaries. A

user may also settle for raw data rather than processed data if the processing resources are too busy. Also in a distributed environment, each application, operating on behalf of a user, may have a different priority than other applications. These priorities may reflect a user's economic situation or military rank.

To formally state the problem any architecture that supports adaptive applications must address, we must first begin by enumerating our notation. Following these definitions and assumption statements, we present several mathematical constraints and an objective function that quantifies the problem. In order to understand how the mathematical terms used in the following paragraphs are related to one other, please refer to Figure 30.

$$Application_j \begin{cases} \text{\underline{is associated with:}} \\ P - \text{priority} \\ \\ D - \text{Data} \begin{cases} T - \text{deadline} \\ \text{\underline{has priority/format pairs:}} \\ (\rho_1, F_1) \\ (\rho_2, F_2) \\ \\ (\rho_m, F_m) \begin{cases} \text{\underline{requires resources:}} \\ R_1 \text{of Resource 1} \\ R_2 \text{of Resource 2} \\ R_n \text{of Resource N} \end{cases} \end{cases} \end{cases}$$

Figure 30. Mapping of mathematical terms to an application.

We assume that for every application j, which requires a collection of data D, there is a time $T_{D,j}$ after which the data is no longer required. For example, a commander may be using a planning tool that requires both logistical information from a database and a video of a target location. The target is to be assaulted at 1400, one hour from now. If the commander receives the data after the assault, it is useless. In this example:

- j is the planning tool,
- D is the video and database information, and

71

- $T_{D,j}$ is 1400.

If the deadline $T_{D,j}$ cannot be met, and application j is written so as to be able to accept different formats of the data, then a different format, $F_{1,D,j}, F_{2,D,j}, ...$ or $F_{m,D,j}$ (where the number of formats, m, depend on D and j), in which the data can be supplied could be sent to the commander. For example:

- $F_{1,D,j}$: full video and database information

- $F_{2,D,j}$: color pictures and database information

- $F_{3,D,j}$: black and white pictures and database information

- $F_{4,D,j}$: database information

When it is obvious which application j and which data item D we are referring to, we will simplify our notation and use only T, F_i, and F_m to refer to the deadline, the ith format, and the last format, respectfully. We associate with each format, F_i, a normalized priority, ρ_i, that reflects the desirability of that format. The sum of the normalized priorities for all of the different formats of the same data is 1. For example, an application may accept data in one of two different formats where their normalized priorities are .9 and .1, meaning that the first format is much preferable to the second. In another case, the different priorities might be each .5, meaning that the application does not care which format it receives.

Next we assume that each application j has a priority P_j, that reflects its importance and that of its user with respect to other applications and their users. Without loss of generality, we will assume that the priorities range between 0 and N, with 0 being the highest priority.

Different amounts/types of resources may be needed to acquire 2 different formats. For example F_1 may require 30 minutes of CPU time, 45MB of temporary hard disk storage and 45 minutes of the network resource at 5 Mbits/second. On the other hand, F_2 will have different requirements, perhaps needing only 15 minutes of CPU time, 5 MBytes of temporary hard disk storage and 20 minutes of a network

72

resource at 2 Mbits/second. We assume that the different formats are **resource-predictable**. By this we mean that it is possible to obtain meaningful distributions that allow us to estimate how much of a given resource will be needed to deliver each format. The variance for these distributions may be small for some resources and applications (both because the underlying distribution has a small variance and because enough good sample data is available to accurately estimate the distribution). For other distributions, the variance might be very large.

During the delivery of format F_i, we track the amount, R_K, of resource K that is used, using a function $M_{(i,j,K)}$. Function $M_{(i,j,K)}$ indicates the amount of resource K that was used in an attempt to deliver format F_i to application j.

For a data item to have been **successfully received**, one of its formats must be received in its entirety before time T. That is, from the standpoint of the application, a format that is only partially received is equivalent to not having received the data at all. For example, if an application receives only half of format F_1, two-thirds of the format F_2, and nothing else, that is equivalent to the application not receiving any data. We therefore define a **characteristic** function. The value of the characteristic function, $X_{i,j}$, is 1 for application j, format i, if F_i has been delivered, in its entirety, to application j, and is 0 otherwise. An application may want to deliver F_2, which would require 10 minutes of CPU time, so $R_{cpu} = 10$. If the entire file is delivered before deadline T, then:

- $M_{2,j,cpu} = 10$ (10 minutes of CPU time used) and
- $X_{2,j} = 1$ (file completely delivered).

Another assumption that we make is that the environment can have multiple modes. For example, two such modes may be **normal** and **critical**. In normal mode, it might be just as important to send ten priority 5 messages (from the soldier in the field) as to send one priority 0 message (from the Commander), whereas in the critical mode the single priority 0 message (from the Commander) might be much more important than any number of priority 5 messages. Therefore, we associate

a function with each mode, call it I_{mode}, that will be applied to priorities P_j. We assume that I_{normal} is the identity function. As this function is domain specific, we do not define it further in this thesis. We only note that our architecture must support applications in domains where such priorities exist.

Given today's extensive use of networks and distributed computing, we must assume that resources are **concurrently shared**. By this we mean that at any point in time, multiple new requests may start to use a particular resource and other already existing requests may need additional use of that resource.

Associated with each resource, K, there is a total amount of that resource that can be used until deadline T, namely $U_{K,T}$. For example, it is estimated that a network is providing 5Mbit/sec of throughput, and this throughput will remain steady for the next 10 minutes. An application has just finished processing a large database request (200 MBytes) and wishes to transmit the results to the requestor. The application has a deadline 8 minutes from the current time. From this data, the following calculations can be made:

- $T = T_0$ (current time) $+ 480$ seconds

- $R_{network} = 200$ MBytes $*8$Mbit/Byte $= 1600$ Mbit total throughput required

- $U_{network,T} = 480$ seconds $* 5$Mbit/sec $= 2400$ Mbit available throughput

Since $R_{network} \leq U_{network,T}$, this application will likely meet its deadline T. Additionally we note that there are resources which are *not* dependent upon T. The quantity of those resources, such as main memory and disk space, that is available does not grow over a period of time. If there is a total of 2GB of disk space, then 3 hours from now there is still 2GB (unless a new hard drive is purchased). This is unlike the network resource above, where longer use gets more work accomplished over time.

Stated more formally, ideally the problem that we wish to solve is, when given n applications, to maximize

$$\sum_{j=1}^{n} I_{mode}(P_j) \sum_{i=1}^{m_j} p_{i,j} X_{i,j}$$

subject to

$$\forall j \sum_{i=1}^{m_j} X_{i,j} \leq 1$$

and

$$\forall k, \sum_{j=1}^{n} \sum_{i=1}^{m_j} M_{i,j,k} \leq U_{K,T} \text{ where T is the max(T) }[1]$$

Finally, not all applications that use the shared resources, such as file servers or networks, will go through our software architecture. This is among our most important assumptions. Though these other applications will not use our software architecture as an interface to the resources that are shared, the very fact that they use those resources means that they will affect the performance of our applications. Our architecture must therefore provide a means to measure such use.

---

[1]We must account for two different types of penalties that are associated with different types of resources. The first is a penalty that will decrease resource availability due to sharing resources, such as the context switches that must occur when switching the task a processor is executing. The second penalty makes a resource's availabilty seem larger than it is physically, such as virtual memory.

VII. PROPOSED ARCHITECTURE SOLUTION

In this chapter we suggest a client-server architecture to support adaptive applications. In particular, our proposed architecture will

- permit the easy integration of scheduling heuristics that attempt to solve the optimization problem that we described in the last chapter,

- provide accurate estimates of resource status to the heuristic algorithms, and

- provide accurate estimates of resources required to compute and/or deliver requested formats.

The latter 2 estimates are required to supply inputs for the scheduling heuristics.

Our proposed architecture solution borrows from existing systems such as Condor [Ref. 11] and SmartNet [Ref. 12], recent research in real time network protocols that use reservations [Ref. 13, 14, 15, 16] (see Appendix H), and protocols that are used to transmit wavelets for virtual reality distribution [Ref. 17].

Our new architecture consists of a client library with which the applications link and possibly replicated servers with which the library functions interact. The library aids the application writer by

- hiding the complications of adaptation,

- performing interactions with the servers, and

- solving the problems associated with detecting the current loads on the shared resources.

The servers permit users of different priority, each requiring different qualities of service, to share the resources as the designers and administrators intended. They also provide a damping effect that prevents rapid oscillations between attempts to deliver different size formats of data when the resources become heavily loaded. Before presenting the details of our architecture, we first present an overview of it.

A. OVERVIEW OF OUR ARCHITECTURE

Our architecture consists of a library with which the client links and several, possibly replicated, possibly integrated, servers as shown in Figure 31. Both adaptive
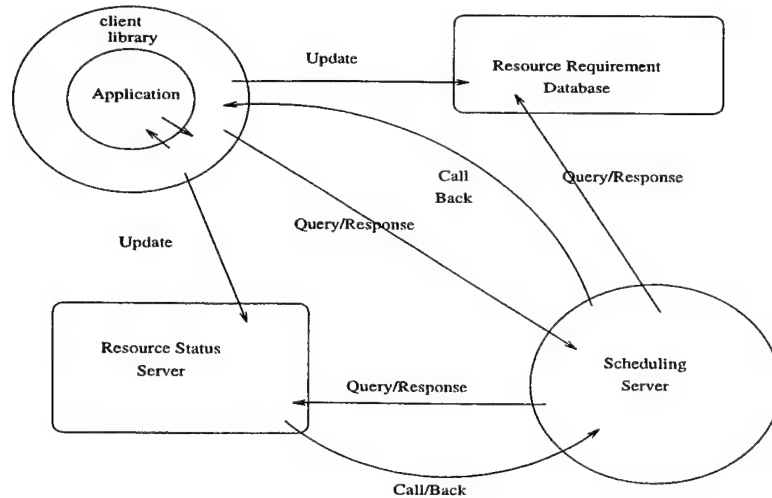
Figure 31. Overview of proposed architecture.

and non-adaptive applications can link with our library, which provides three different functionalities: (1) to transparently attempt to deliver the best possible format for each adaptive application that links with it, (2) to determine the resource needs of applications with which it is linked, and (3) to attempt to ascertain the current load on the various resources that it may need to use.

Below, we describe three different servers because there are three different functionalities that must be embodied in the servers. However, it is conceivable that in different installations the different services might be combined into a single physical server, or that some of the services might be replicated while others are not.

The purpose of the client library that is linked with both adaptive, as well as non-adaptive, applications is to provide an easy-to-use interface to all of the services. The job of the Resource Status Server is to maintain a quickly changing database of estimated loads on the various resources. The Resource Requirement Database helps determine what resources are required to deliver/calculate the different formats. The final server, the Scheduling Server, helps to arbitrate the use of the various resources

78

by the different clients in an attempt to maximize the optimization criteria described at the end of Chapter VI, subject to the constraints, also enumerated there. It is beyond the scope of this thesis to consider which scheduling heuristics may produce the best schedule; this thesis concentrates instead on ensuring that the information needed for executing those scheduling heuristics is supplied by the remaining components of our architecture. We now describe each of the components in some detail. Their detailed design and implementation are currently underway.

B.   THE CLIENT LIBRARY

One of the purposes of the client library is to transparently attempt, with the help of the various servers, to deliver the best possible format for each application that links with it. That is, an application should only be required to furnish a list of formats in which data from a task is acceptable, a normalized priority for each format, and an optional time after which the data is no longer required. The client application should receive either a success or failure response from the library when this function call finishes. A successful response includes both an acknowledgment of the complete acquisition of one of the formats and an indication to the application concerning which of the formats succeeded. A failure indicates that none of the formats was able to be acquired[1].

Upon reception of a request from an application to obtain one of the given set of formats, the client library will contact the Scheduling Server to determine which of the formats it should attempt to acquire. After receiving a response from the Scheduling Server, the client library, on behalf of the client application, will attempt to acquire the specified format. If the status of the resources changes dramatically, the Scheduling Server will issue a call back to the client library indicating that another format would be more appropriate. Additionally, if the client library perceives that the mix of resources required to obtain the format is substantially different from that

---

[1] A failure to acquire any format of the data by a given deadline is also considered a failure.

predicted using the Resource Requirement Database, or it perceives that the load on a resource is substantially different from that estimated by the Resource Status Server, it will notify the Scheduling Server (which may later call back with a request to acquire a different format).

Another purpose of the client library, when linked with either an adaptive or a non-adaptive client, is to ascertain both the resource requirements of various tasks and perceived loads on those resources. In order to estimate these quantities, our architecture will wrap system calls in some light weight code that estimates both of these quantities and periodically updates both the Resource Requirement Database and the Resource Status Server. We currently intend to implement the system call wraps using the Synthetix Toolkit available from the Oregon Graduate Institute [Ref. 18] so that very little overhead will be added to the call.

C.   RESOURCE STATUS SERVER

The job of the Resource Status Server(s) is to maintain a quickly changing database of estimated loads on the various resources. In our distributed architecture, we have one Resource Status Server responsible for each shared resource, or, in some situations, a single server may be responsible for an entire set of shared resources. This server is passive, that is, it does not actively use the resources for which it is responsible. We made this design decision because of the incredible load (overhead) that an active server would place on some shared resources. For example, some resources, such as networks, are extremely hard to actively monitor, requiring a high rate of sampling. Such a substantial stress on a resource might prevent any real work from being accomplished by applications that use the resource. This can be particularly harmful if the system is already in a critical mode (extremely busy). Instead, our passive server collects information from the client libraries. As the client libraries use the resources to do actual work, relevant statistics are sent to the appropriate Resource Status Server, requiring very little overhead.

80

In addition to maintaining this repository of information, the Resource Status Server(s) answer queries from the Scheduling Server. When a client library makes a request to obtain one of several formats to the Scheduling Server, the Scheduling Server queries the various Resource Status Server(s) to determine the loads on the resources that might be useful in obtaining each format.

A typical sequence of events that might be executed by a Resource Status Server follows. First the Scheduling Server, operating on behalf of a client application, will use `QueryStatus(Resource)` to obtain an initial estimate of the load that particular resource is experiencing (Figure 32).
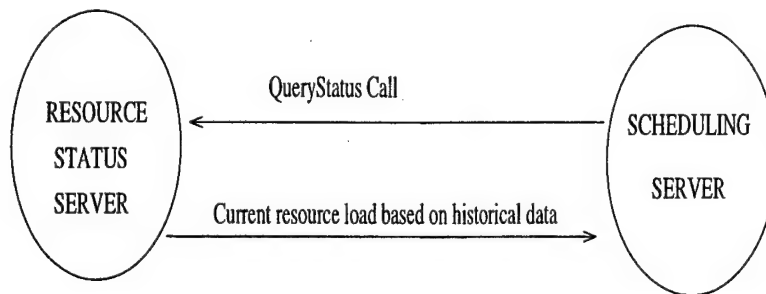
Figure 32. Resource Status Server receiving `QueryStatus()` call.

Once a scheduling decision is made, the Scheduling Server informs the Resource Status Server(s) of the additional loads that it expects the client application to place on the various resources. During execution of the work, the client library will periodically update the server on its experience with each particular resource, as shown in Figure 33. The call used here will be `UpdateServer(ResourceStatus)`.
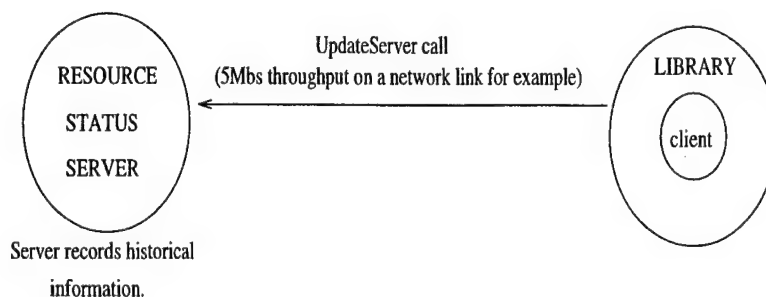
Figure 33. Resource server receiving `UpdateServer()` call.

81

Because the Resource Status Server(s) are informed by many clients concerning the status of the resources, our servers can detect when different modes, such as **normal** and **critical** have been entered. Our servers use a stochastic generalization of the Kalman Filter [Ref. 19] to determine the mode. They can then adjust the priorities of different requests based upon this mode and can alter the advice given to their clients when they detect a change in the mode.

We note that our architecture does not rely upon all of the users of a resource to inform that resource's status server of their use. In fact, part of the job of the client library is to dynamically detect how much of each resource remains available for allocation and to inform the Resource Status Server of any changes.

D.   RESOURCE REQUIREMENT DATABASE

Much of the Resource Requirement Database is modeled after SmartNet's wall clock time database [Ref. 20] that keeps track of the amount of time required to execute compute-intensive tasks on various high performance computers. However, our proposed Resource Requirement Database maintains more fine granularity data, such as the amount of main memory and cache required for efficient execution, as well as the amount of file service needed. Like the Resource Status Server, this database is updated by the client libraries and queried by the Scheduling Server. Like the SmartNet wall clock time database, this database leverages the advantage of using Compute Characteristics [Ref. 21]. This database is updated when tasks are completed.

E.   THE SCHEDULING SERVER

The Scheduling Server contains many heuristics that can be useful in scheduling prioritized tasks in a heterogeneous environment. The Client Library contacts the Scheduling Server to determine which of a variety of formats of data it should attempt to acquire. The Scheduling Server first queries the Resource Requirement Database to

determine an estimate of the resources required to acquire and/or compute each of the different formats and then contacts the Resource Status Server to determine the load on each of the resources that might possibly be used to obtain any of the formats. Based upon other activity and the various priorities in the system, the Scheduling Server then responds to the client library to indicate: which format the client library should attempt to acquire, which resources it should use to acquire that format, and the current estimate of the loads on each of those resources. It then notifies the Resource Status Server of the load that the client is expected to place on these resources.

If the Scheduling Server should later receive a call back from the Resource Status Server, indicating a substantial change in the load on a resource that a client is using, it will re-calculate the schedule to determine whether any client should acquire a different format from that which it is currently acquiring. If it determines that a change is required, the Scheduling Server will then issue a call back to the client library.

In addition to possibly receiving a call back from the Resource Status Server, the Scheduling Server may receive a notification from the client library indicating either that the load on a resource was substantially different from predicted or that the resource requirements were substantially different from the predictions. In this case, also, the Scheduling Server may need to re-calculate and disseminate new schedules to various clients.

F.   PRIORITY MODELS AND ECONOMIC MODELS

Finally, we note the difference between our priority model, the model of choice for both military and internal corporate use, and an economic model, likely the model of choice for commercial users and therefore, likely, the direction that COTS applications will go. An economic model can coexist with our priority model using something similar to Figure 34. The major difference between the two-tiered priority model that we have already described, wherein an application has a priority and it prioritizes its

own demand, and an economic model, where different applications have a different amount of money and are willing to pay a different price for different qualities of service, is *replenishment.* In our model, we do not need to be concerned with whether a client is replenished. If an economic model were to be used on top of what we have already described, an additional library to establish priorities, based upon a budget, would have to be developed. As the application spends its budget, the priority it could purchase would decrease until its budget is replenished.
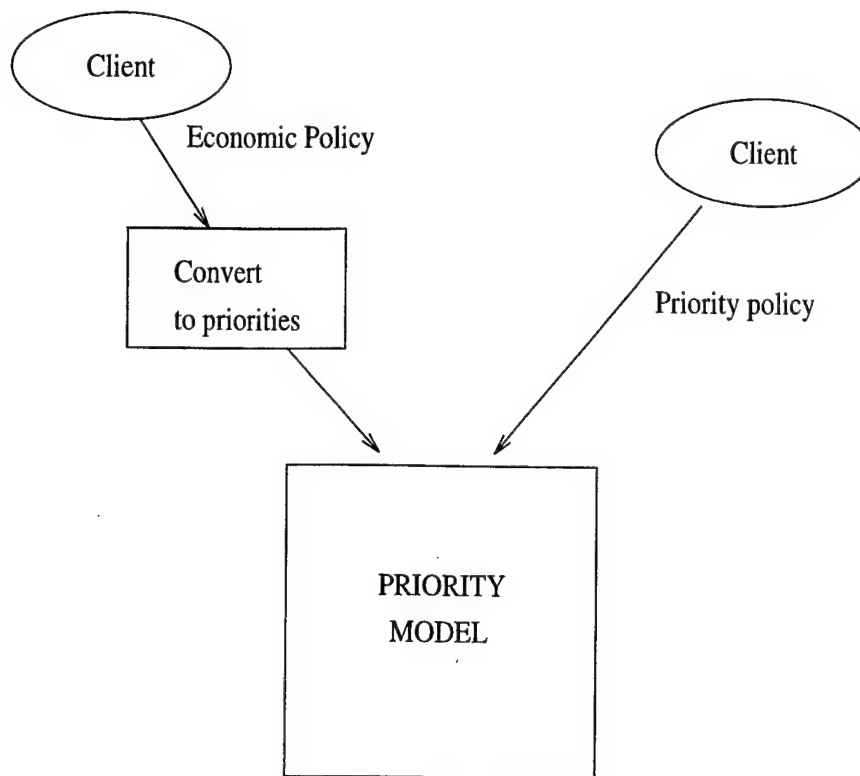
Figure 34. Using an economic model with our priority model.

VIII. SUMMARY

An adaptive application is one that can accept or send multiple forms of the same data, and chooses which to send based upon resource loads. For example, if a video file is too large to reach its destination by a given time, due to a heavily loaded compute environment, then an application may choose to send black and white pictures instead. In our applications, if the data is not delivered by its deadline, the data becomes useless. In order to correctly decide which format to send, however, an application must know what the load is on all the resources it intends to use.

Developing an *adaptive* application that automatically schedules VTC sessions and disseminates conference information requires that the application have accurate knowledge of the loads on the resources that it uses. The current implementation of the JTF Architecture provides a Communications Server that estimates the load of the network resource. Our original intent was to use this server when delivering data such as video files, database information, and large text files to participants in a VTC.

A. SUMMARY OF OUR EARLY EXPERIENCES

In order to become familiar with the structure and the objects of the JTF Reference Architecture, we initially wrote many test programs. With our extensive knowledge of C++ programming, it took us only a month to become familiar with the entire JTF Reference Architecture's structure of services. The reuse of objects was easy, as was described in the use of the Trigger and Worklow objects in Chapter III. We encountered problems with only the Communications Server.

In order to learn the capabilities of the currently implemented JTF ATD Communications Server, we ran several tests and experimented with many of its functions that are identified in its Interface Definition Language. Initially we had several problems gaining access to the Communications Server object. Several weeks were spent configuring scripts to automatically start the Communications Server correctly in the

CORBA environment. Based on these problems, we have concluded that the CORBA environment is very administratively intense, and that this situation needs to be alleviated if the software is to be used in a crisis situation.

We soon discovered that the CS_CommServer_RequestQoS function was the only one that returned anything back to the client, and focused the rest of our experiments on it. Our initial approach was to pass the function different Quality of Service (QoS) parameters, repeatedly requesting additional throughput. Unfortunately, the Communications Server never reacted to these changing input parameters. After discussing this with the Communications Server's implementors, we discovered that the input parameters were not currently used to calculate the QoS for the network resource.

We then ran several test programs to determine how accurately the raw measurements (that the Communications Server was returning) were at predicting the actual load on the network. Unfortunately, our results showed that the values did not accurately reflect the load that we were placing on the network. There are several reasons why the current Communications Server implementation does not completely reflect the load on the network. First, the values that are returned are instantaneously collected. Due to the bursty nature of network traffic, the Communications Server was sometimes reporting the statistics at a busy instant in time, when, in reality, the *average* load on the network was light. Also, the opposite occurred, wherein a low point was sampled when the network was in fact quite busy. Unfortunately, the Communications Server coupled this problem with not keeping a history of recent sample information. Second, even network-intensive applications use additional resources besides the network, such as CPU, memory, and hard drives, whose loads must also be considered. Based both on these discoveries and our simulation results, we developed a proposed client-server architecture that should provide adaptive applications with a better estimate of resource loads.

B. SIMULATIONS DETERMINING SERVER ACCURACY

We ran several simulations that modeled only the network resource. We simulated two different modes, one when the network was heavily loaded (crisis situation), and one where the network was more lightly loaded (normal use). Among other statistics, we recorded the number of messages that did not arrive before their specified deadline.

In crisis mode, our results showed that a resource server that could accurately estimate the actual network throughput within 5 Kbits/second would always deliver at least 96% of the data before its deadline. On the other hand, an intrusive server, such as the Communications Server, had a failure rate between 14% and 30%. We note that the adaptive applications that used the Communications Server did well compared to those that did not adapt at all, which produced a 98% failure rate.

C. A PROPOSED ARCHITECTURE TO SUPPORT AD-APTIVE APPLICATIONS

Our proposed architecture is generic in that it attempts to accurately estimate the loads on many different types of resources so that it can effectively support adaptive applications. Our proposed architecture uses a library of functions that execute on behalf of the application. The application initiates a call to the library, telling the library what type of service it requires. The library then performs the work of adapting to resource loads and performing, in some form, the work requested. Once the work is completed, the library informs the application how successful it was and passes any requested data to the application.

Our library communicates with a collection of servers to gain an estimate of the current load on the resources it wishes to use. Once the library receives these estimates, a Scheduling Server calculates which forms of data the client application should be able to successfully calculate and/or acquire given the specific deadline of that application. As the library sends or receives the data, it periodically determines

the QoS it is receiving and updates the appropriate Resource Status Servers.

Our Resource Status Servers are not intrusive. They receive resource load estimates from the libraries that are actually using the resources. In addition to tracking the loads on resources, a Scheduling Server helps determine the proper allocation of resources. Based on the priority of an application, it may receive greater use of a resource in a crisis situation if it has a high priority, or be asked to reduce its resource use if it has a low priority.

In order to accurately schedule resources for particular tasks, the Resource Requirement Database communicates resource needs of the tasks to the Scheduling Server. The Resource Requirement Database receives updates from the client library on the specific amount of memory, cache, and other resources that a task used. When a client application completes a task, this information is added to the database to provide better resource use estimates for scheduling the task in the future.

D. CONCLUSIONS AND FUTURE WORK

In a military environment, handling the crisis situation is vital. In this thesis, we show through simulation that, in a crisis situation, very accurate resource loading information is required to permit applications to adapt and, in particular, to ensure that the highest priority applications receive sufficient resources. We propose an architecture to support adaptive applications in these situations.

Future work is required to continue to refine our proposed architecture. This refinement will require prototype implementations of both our client library and our proposed servers. Additional simulations should be run using more nodes, network routing, varying the weights on how instantaneous readings are recorded, and then incorporating other resources such as CPUs, memory, and hard drives. To ensure that applications which will be useful in crisis military situations can be built from COTS software, a mapping from an economic model to a priority based model will also be needed.

APPENDIX A. C++ CODE FOR THE VTCAGENT

```cpp
//------------------------vtcagent.C--------------
//This is the main program that is starts the vtcagent
//-----------------------------------------------

#include "util.h"

//This trigger is activated when a new tasker is assigned to the user
void c2WorkflowUser::trigger(char* reason, c2NameValues_var& info) {
    C2SchemaObject::trigger(reason, info);

    for (int i=0; i < this->taskers._length; i++) {
        if ((strcmp(user->taskers[counter]->what->headers[2],TASK_TYPE)) == 0) {
            actOnVTC(user->taskers[counter]->what,user);
        }
    }
}




int main (int argc, char **argv)
{
    char *user_name = NULL;

    if (argc > 2) {
        cerr << "Usage:\n\t" << argv[0] << " [user@host]\n";
        return 1;
    }
    if (argc > 1) {
        user_name = argv[1];
    }

    XtAppContext app;
    XtAppInitialize(&app, "MyAppClass", NULL, 0, &argc, argv, NULL, NULL, 0);

    X_FineC2_Trigger_API trigger(app);  //this is what binds to the
                                        //Socket_Trigger_Server

    finec2_default_trigger = & trigger.info; //from finec2.h
```

```cpp
      //this binds to the Workflow Server
      c2WorkflowDirectory::get(2, &trigger);

      c2WorkflowIdentity *user = c2WorkflowUser::my_identity(user_name);

      char *Dir = new char[100];
      Dir = getenv("HOME");

      strcat(Dir,VTC_DIR);
      //checks to see if directory is there, if not make it.
      checkVTCDir(Dir);


      //go through all user's taskers to see if it is a VTC and we need to act
      for (int counter = 0; counter < user->taskers._length; counter++) {

          if ((strcmp(user->taskers[counter]->what->headers[2],TASK_TYPE)) == 0) {
              actOnVTC(user->taskers[counter]->what,user);
          }

      }


      //wait for something to happen
      XtAppMainLoop(app);
      return 0;

}



//------------------------util.C-------------
//This file contains utility functions that are part
//of the vtcagent
//-------------------------------------------------

#include "util.h"

string_hash sessions;             //holds VTC taskers that we know about
char *User_Dir= new char[100]; //holds the name of user's local VTC directory
```

```c
//checks to see if a directory exists, if not it makes it.  The first
//time through is when we get the user's VTC directory.
//From then on, we don't have to do this.
//Directory is made if it does not exist with the Dir name that is passed in
//Returns 1 if we make a directory, 0 if it already existed.
int checkVTCDir(FineC2_string Dir)
{
    static int counter = 1;
    int status;
    struct stat statbuf;

    //First time through, get user directory
    if (counter == 1) {
        strcpy(User_Dir, Dir);
        counter++;
    }

    //Check to see if VTC directory exists, if not make it and notify
    if ((status = stat(Dir, &statbuf))) {
        mkdir(Dir,0777);
    }

    return status;
}


//if the tasker has not been seen yet, add it to the hash table
//This function make it quicker to look up, instead of a linked list
//returns a 1 if we insert a new one, a 0 if it was already here.
int addToHash(c2WorkflowTasker *tasker, FineC2_string &name)
{
    int status = 0;

    if (!sessions.lookup(name)) {
        sessions.set(finec2_strdup(name), (char*)tasker);
        status = 1;
    }

    return status;

}
```

```cpp
//creates a session name for the TASK_TYPE being tracked
//This is a unique name based on the originator and subject/date
void getSessionName(c2WorkflowTasker *tasker, FineC2_string &name)
{
    FineC2_string temp;
    int length,templen, counter;


    if (strlen(tasker->headers[0]) == 0) {
        length = strlen(tasker->tasked_by->print_name) +
                strlen(tasker->headers[1]) + 2; //add the "." and the "\0" = 2
        name = new FineC2_char[length];
        strcpy(name,tasker->tasked_by->print_name);
        strcat(name,".");
        temp = new FineC2_char[strlen(tasker->headers[1])+1];

        //replace spaces with uderscores
        for (counter = 0; counter < strlen(tasker->headers[1]);counter++) {
    if (*(tasker->headers[1]+ counter) == ' ') {
                temp[counter] = '_';
            }
            else {
                temp[counter] = *(tasker->headers[1]+ counter);
            }
        }

        strcat(name,temp);
    }
    else {
        length = strlen(tasker->tasked_by->print_name) +
                strlen(tasker->headers[0]) + 2; //add the "." and the "\0" = 2
        name = new FineC2_char[length];
        strcpy(name,tasker->tasked_by->print_name);
        strcat(name,".");
        temp = new FineC2_char[strlen(tasker->headers[0])+1];

        //replace spaces with uderscores
        for (counter = 0; counter < strlen(tasker->headers[0]);counter++) {
    if (*(tasker->headers[0]+ counter) == ' ') {
                temp[counter] = '_';
            }
            else {
```

```
                temp[counter] = *(tasker->headers[0]+ counter);
            }
        }
        temp[counter] = '\0';
        strcat(name,temp);
        delete temp;
    }
}



//if there are any attachments, exec a process that will distribute the files
void distribFiles(c2WorkflowTasker *tasker,char *responsible)
{
    char *host;
    char *reference;
    pid_t parent,child;       //process ids for the parent and child


    host = strtok(responsible,"@");
    host = strtok(NULL,"@");

    //if responsible does not have host, use the one from this host
    if (host == NULL) {
      strcpy(responsible,"jtfweb5");  //fix this for any host
       host = responsible;
    }

    //if there are attachments, distribute them
    if (tasker->attachments._length > 0) {

       for (int ix = 0; ix < tasker->attachments._length; ix++) {
          CORBA::String_var oref =
            tasker->attachments[ix]->object->corba_objref->_object_to_string();
          reference = new char[strlen(oref)+1];
          strcpy(reference,oref);

          parent = getpid();
          child = fork();
          if (child == 0) {
          execl("mysend","mysend",host,reference,tasker->attachments[ix]->name,
                (char*)0);
```

93

```
            }
            delete reference;
        }

    }

  return;

}


//Create a status file for this tasker, tracking which files have been
//ditributed and any other needed info.  Places it in the users
//Planner/VTC Directory
void initOwnerFile(FineC2_string &filename,c2WorkflowTasker *tasker)
{
    int status, length;
    struct stat statbuf;
    FineC2_string tempPath = NULL;
    ofstream session_file;

    length = strlen(filename) + strlen(User_Dir) + 2;
    tempPath = new char[length];
    strcpy(tempPath,User_Dir);
    strcat(tempPath,"/");
    strcat(tempPath,filename);

    //if it is not there, make it
    if ((status = stat(tempPath, &statbuf))) {
       session_file.open(tempPath, ios::out);
       for (int ix = 0; ix < tasker->attachments._length; ix++) {
 session_file<<tasker->attachments[ix]->name<<"   0"<<endl;
       }
     //need to put stuff in it to initialize add function to place attachments
     //place #attachments, etc....
     session_file.close();
    }

}
```

```
//When a VTC task is found, figure out what to do with it
//If it is a new one, then make directory and add time to list.
//If this user is the originator, distribute attachments, else,
//get ready to receive
//files if there are any.
void actOnVTC(c2WorkflowTasker *tasker, c2WorkflowIdentity *user)
{
    FineC2_string session_name = NULL;   //holds the session name
    FineC2_string status_file = NULL;    //holds name of status file
    int length, counter = 1;

    getSessionName(tasker,session_name);

    length = strlen(session_name)+2;
    status_file = new char[length];
    FineC2_string responsible =
            new char[strlen(tasker->responsible->print_name)+1];
    strcpy(responsible,tasker->responsible->print_name);
    status_file[0] = '.';

    for (counter; counter < length;counter++) {
        status_file[counter] = *(session_name + counter-1);
    }
    status_file[counter] = '\0';

    //add to a hash table, quick to look up for future reference
    if (addToHash(tasker,session_name)) {
        if (strcmp(tasker->tasked_by->print_name,user->print_name)==0) {
            initOwnerFile(status_file,tasker);
            distribFiles(tasker,responsible);

        }

    delete [] session_name;
}
```

APPENDIX B. DIFFICULTIES EXPERIENCED WHILE ACCESSING THE COMMUNICATIONS SERVER

When we initially attempted to run the example code presented in Chapter IV, we received an error that told us that we had no permission to use the CommServer object. After exchanging several emails with BBN, we discovered that the CORBA settings for the Communications Server were set to:

```
Server details for server : CommServer

        Comms         : tcp
        Code          : xdr
        Activation    : shared
        Owner         : root
        Launch        : ;
        Invoke        : ;
```

These settings showed that no client could *launch* or *invoke* the Communications Server. After the system administrator changed the settings for Launch and Invoke to "all," we could run our test program with success.

When we ran our simple test program from above, we received a response from the Communications Server. Here is some sample output:

```
[4720: New Connection (jtfweb5,IT_daemon,*,root,pid=258,optimised) ]
[4720: New Connection (jtfweb5,CommServer,*,jpkresho,pid=25928,optimised) ]
```

From the above output, we see that CORBA found the CommServer object on the jtfweb5 computer, along with the user that invoked it (jpkresho in this case).

Unfortunately, after this initial successful access to the Communications Server, there still have been many days in which we could not access it. Whenever the compute server is rebooted, several configuration scripts must be run by an administrator in order for us to access the Communications Server again. These scripts start the server and gather initial data.

APPENDIX C. C++ CODE FOR FIRST SET OF FILE TRANSFER TESTS

```
//------------------------sendutils.h----------------------

#include <sys/types.h>
#include <sys/socket.h>
#include <sys/errno.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>
#include <iostream.h>
#include <unistd.h>
#include <fcntl.h>

#define PORT_MAX 49

extern "C" int gethostname(char *name, int namelen);

//gets the port for the data connection
int get_port(char *,char *,int);


//sends command over a socket
int sendcmd(int, char*, char*);


//gets reply from a socket
int getreply(int, char*);


//get the IP address and the port number, then put it into a string
void getThisIP(char *charPtr,char *, char *,char *thisIP);


//------------------------sendutils.C----------------------
//Utility functions to help ftpsend.C do it job
//--------------------------------------------------------
```

```c
#include "sendutils.h"

//gets the port for the data connection
int get_port(char *port_save1,char *port_save2,int socknum)
{
    char reply[256];
    char *cp, *start, digit[2];
    int num, totlen, commas=0,ps1=0,ps2=0;

    if (write(socknum,"PASV\n",5) < 0) {
        perror("write()");
        return (0);
    }

    if (read(socknum,reply,256) < 0) {
        perror("read()");
        return (0);
    }

    // Position cp on first digit
    for (cp = &reply[5]; (*cp) && !isdigit(*cp); cp++ ) ;
    start = cp;
    num = 0;
    for (totlen=0 ; ; cp++, totlen++) {
if ((! *cp) || *cp == ')' || *cp == ',' || *cp == ' ') {
    if (num > 255)
        return(0);
    else {
        if (*cp == ',') {
                    commas++;
                }

        num = 0;
            }
}
else {
    //save the first 8 bits of the port
            if (commas == 4) {
                port_save1[ps1] = *cp;
                ps1++;
            }
    }
```

```c
                //save the second 8 bits of the port
                if (commas == 5) {
                    port_save2[ps2] = *cp;
                    ps2++;
                }

        strncpy(digit, cp, 1);
        digit[1] = '\0';
        if (isdigit(digit[0]))
            num = (num*10) + atoi(digit);
        else
            return(0);
        }
if ((! *cp) || *cp == ')'){
                port_save1[ps1]='\0';
                port_save2[ps2]='\0';
        break;
            }
        }
        if ((totlen==0) || (totlen >= PORT_MAX))
            return(0);
        else {
            return(1);
        }
}



//the command must come in as "command %s\n", then command is sent to
//distant end
int sendcmd(int socknum, char *command, char *arg)
{
    char cmdbuff[256];

    sprintf(cmdbuff,command,arg);

    if (write(socknum,cmdbuff,strlen(cmdbuff)) < 0) {
        perror("write()");
        return 0;
    }

    return 1;
```

```
}


//this is used just to get the reply out of the buffer for the socket
int getreply(int socknum, char *replybuf)
{

    if (read(socknum,replybuf,1024) < 0 ) {
        perror("read()");
        return -1;
    }

    return 1;
}



//takes strings which contain an IP address and port combination and
//combines them into a single string for output in the PORT commmand
void getThisIP(char *charPtr,char *port1,char *port2,char *thisIP)
{

#define UC(b)     (((int)b)&0xff)

    sprintf(thisIP,"%d,%d,%d,%d,%s,%s", UC(charPtr[0]),UC(charPtr[1]),
            UC(charPtr[2]),UC(charPtr[3]),port1,port2);

}



//------------------------ftpsend.C------------------------
//Connects to ftp port and transfers files from one machine to another
//This version reads the file from disk each time it is transferred
//-----------------------------------------------------------

#include "sendutils.h"

#define PORTNO  21         //well known ftp port for command connection
#define USER "user"        //this is where a login name would go
#define PASS "!@#*)^"      //password for this user, a little dangerous

main(int argc, char **argv)
{
```

```cpp
    int socknum;              //socket for the command connection
    pid_t parent,child;       //process ids for the parent and child (data server)
    struct hostent *hp;       //holds the information of the remote host
    char buf[1024];
    struct in_addr remote_address; //used to connect to the remote host
    struct in_addr this_address;   //used for the address of this host
    char port_save1[4], //first 8 bits of port
         port_save2[4]; //second 8 bits of port;

    union sock
    {
        struct sockaddr s;
        struct sockaddr_in i;
    }sock;

    if (argc < 3) {
      cerr<<"YOU NEED at least 2 arguments:  mysend destination file"<<endl;
      return -1;
    }


    //now setup the socket for connecting to the ftp port
    socknum = socket(AF_INET, SOCK_STREAM, 0);
    hp = gethostbyname(argv[1]);
    strncpy((char *)&remote_address,hp->h_addr_list[0],sizeof(in_addr));
    sock.i.sin_family = AF_INET;
    sock.i.sin_port = htons(PORTNO);
    sock.i.sin_addr = remote_address;

    //connect to the socket and pass commands/get replys
    if (connect(socknum, &sock.s, sizeof(sockaddr)) < 0) {
        perror("connecting stream socket");
        return(-1);
    }

    if (!getreply(socknum,buf)) {
        return(-1);
    }

    //send user name
    if (!sendcmd(socknum, "user %s\n", USER)) {
        return(-1);
```

103

```
}

if (!getreply(socknum,buf)) {
    return(-1);
}

//now the password
if (!sendcmd(socknum, "pass %s\n", PASS)) {
    return(-1);
}

if (!getreply(socknum,buf)) {
    return(-1);
}

//change to binary format for file transfer
if (!sendcmd(socknum, "type %s\n", "i")) {
    return(-1);
}

if (!getreply(socknum,buf)) {
    return(-1);
}
//now get IP address for this host for the data connection
char thisHost[50];
gethostname(thisHost,50);
hp = gethostbyname(thisHost);
strncpy((char*)&this_address,hp->h_addr_list[0], sizeof(in_addr));
sock.i.sin_addr = this_address;

char thisIP[30];
char *charPtr = (char*)&sock.i.sin_addr;

//loop for a number of time, currently set to 5, but can be very large
//in order to sustain traffic on the net
for (int jx = 1; jx <=5; jx++) {

    //for each file from the command line, copy to the destination
    for (int ix = 2; ix < argc; ix++) {

        //ask the remote machine for a port
        port_save1[0]='\0';
```

```
port_save2[0]='\0';
get_port(port_save1,port_save2,socknum);
getThisIP(charPtr,port_save1,port_save2,thisIP);

//now the PORT command
if (!sendcmd(socknum, "port %s\n", thisIP) < 0) {
   return(-1);
}

if (!getreply(socknum,buf)) {
   return(-1);
}

parent = getpid();
child = fork();

//if the child, exec the server to accept the data connection
if (child == 0) {
   execl("dataconn","dataconn",port_save1,port_save2,argv[ix],(char*)0);
}

int goid = open("go.txt",O_RDONLY);
//wait until the data connection is ready before going on
while (goid < 0) {
   sleep(1);
   goid = open("go.txt",O_RDONLY);
}

close(goid);

//now the stor
if (!sendcmd(socknum, "stor %s\n", argv[ix]) < 0) {
   return(-1);
}

if (!getreply(socknum,buf)) {
   return(-1);
}

int doneId = open("done.txt",O_RDONLY);
//wait unitl data is transferred before moving on
while (doneId < 0) {
```

```
                sleep(1);
                doneId = open("done.txt",O_RDONLY);
            }

            close(doneId);
            system("rm done.txt");

            if (!getreply(socknum,buf)) {
                return(-1);
            }

            if (!sendcmd(socknum, "dele %s\n", argv[ix]) < 0) {
                return(-1);
            }

            if (!getreply(socknum,buf)) {
                return(-1);
            }
        }//end inside for, individual file
    }//end of outside for, completed one set of files

    close(socknum);
    return 0;

}


//----------------------server.h----------------------

#include <sys/types.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/errno.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <string.h>
#include <iostream.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/timeb.h>
```

```cpp
//takes two strings and converts them into a port number (integer)
int get_port(char *, char *,int &);

extern "C" int gethostname(char *name, int namelen);

extern "C" int ftime(struct timeb *tp);


//------------------------dataconn.C------------------------
//Sends the actual data to the distant machine using the
//data connection established by ftpsend.C
//----------------------------------------------------------
#include "server.h"

const int NAME_SIZE = 50;


main(int argc, char **argv)
{
    int goId;
    int datanum, portnum=0;
    int length = NAME_SIZE;
    struct hostent *hp;
    char buf[1024];
    char thisHost[NAME_SIZE];
    struct in_addr this_address, accept_addr;

    union sock
    {
        struct sockaddr s;
        struct sockaddr_in i;
    }sock;

    get_port(argv[1],argv[2],portnum);

    gethostname(thisHost,length);
    hp = gethostbyname(thisHost);
    strncpy((char*)&this_address,hp->h_addr_list[0], sizeof(in_addr));
    sock.i.sin_port = htons(portnum);
    sock.i.sin_addr = this_address;
```

```cpp
datanum = socket(AF_INET, SOCK_STREAM, 0);

//bind to a port on this machine
if (bind(datanum, &sock.s, sizeof(sockaddr)) < 0) {
   perror("in the bind");
   return(-1);
}

//now do the listen and wait for a connection
if (listen(datanum,1) < 0) {
   perror("in the listen");
   return -1;
}
else {
   //tell ftpsend.C it is OK to send the stor command
   goId = creat("go.txt",0777);
}


int addr_len = sizeof(sockaddr);

//open the data connection when the other side attempts to connect
int test = accept(datanum,&sock.s,&addr_len);

if (test < 0) {
   perror("in the accept");
   return -1;
}
else {
   close(goId);
   system("rm go.txt");
}

int fileid;
struct stat fileStat;
stat(argv[3],&fileStat);
char *file_buf = new char[fileStat.st_size+1];

fileid = open(argv[3],O_RDONLY);
read(fileid,file_buf,fileStat.st_size);
```

```cpp
    timeb *startTime = new timeb;
    ftime(startTime);   //start the timer

    cout<<"This was the START time of "<<argv[3]<<": "<<startTime->time<<endl;

    //send the file to the distant machine
    if (write(test,file_buf,fileStat.st_size ) < 0) {
       cerr<<"Error in write"<<endl;
       retrun -1;
    }
    else {
      ftime(startTime);
      cout<<"This was the FINISH timeof "<<argv[3]<<": "<<startTime->time<<endl;
      close(datanum);
      close(test);
      int doneId = creat("done.txt",0777);
    }

    return 0;
}
```

APPENDIX D. C++ CODE FOR SECOND SET OF FILE TRANSFER TESTS

```
//-----------------------side1.C----------------------
//Connects to the other side, then transfers a file to the other side,
//and receives it back, displaying the time it took for a roundtrip of the
//file.  This conitnues until user interruption
//----------------------------------------------------


#include <sys/types.h>
#include <sys/stat.h>
#include <sys/socket.h>
#include <sys/errno.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>
#include <iostream.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/timeb.h>
#include <assert.h>



extern "C" int gethostname(char *name, int namelen);

extern "C" int ftime(struct timeb *tp);

main(int argc, char **argv)
{
    int socknum;
    struct hostent *hp;       //holds the information of the remote host
    struct in_addr remote_address; //used to connect to the remote host
    double time1,   //start time in seconds using decimals for milli seconds
           time2;   //end time in seconds using decimals for milli seconds

    union sock
    {
        struct sockaddr s;
```

```cpp
    struct sockaddr_in i;
}sock;


if (argc != 4) {
  cerr<<"YOU NEED  3 arguments:  side1 destination file port"<<endl;
  return 0;
}


int len = 0;
int total = 0,counter =1;
timeb *startTime = new timeb;
timeb *stopTime = new timeb;


int portnum = atoi(argv[3]);


//now setup the socket for connecting to the transfer port
socknum = socket(AF_INET, SOCK_STREAM, 0);
hp = gethostbyname(argv[1]);
strncpy((char *)&remote_address,hp->h_addr_list[0],sizeof(in_addr));
sock.i.sin_family = AF_INET;
sock.i.sin_port = htons(portnum);
sock.i.sin_addr = remote_address;


//connect to side2
if (connect(socknum, &sock.s, sizeof(sockaddr)) < 0) {
   perror("connecting stream socket");
   return(1);
}


char *temp_buf = new char[10000];


int fileid;
struct stat fileStat;
stat(argv[2],&fileStat);
char *file_buf = new char[fileStat.st_size+1];
assert(file_buf != NULL);


fileid = open(argv[2],O_RDONLY);


//read the entire file into a buffer
read(fileid,file_buf,fileStat.st_size);
```

```
        close(fileid);

        //now loop for a while, until user interrupts
        while (total==0) {

            ftime(startTime);    //start the timer

            //transfer the file to the other side
            if ((len=write(socknum,file_buf,fileStat.st_size )) < 0) {
                perror("Error in write");
            }

            //now receive the same file back from the other side
            while ((len=read(socknum,temp_buf,10000 )) > 0) {
                total += len;
                if (total == fileStat.st_size) {
                    ftime(stopTime);   //end of round trip
                    time1 = startTime->time + (startTime->millitm/1000.0);
                    time2 = stopTime->time + (stopTime->millitm/1000.0);
                    cout<<"Round trip #"<<counter<<"time was: "
                    <<time2-time1<<endl;
                    break;
                }
            }
            counter++;
            total = 0;
        }
        close(socknum);

        return 1;
}



//------------------------side2.C----------------------
//Initally waits on a port until side1 connects.  It then
//receives a file from side1 and sends it back.
//-------------------------------------------------------

#include <sys/types.h>
#include <sys/stat.h>
#include <sys/socket.h>
```

```cpp
#include <sys/errno.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>
#include <iostream.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/timeb.h>
#include <assert.h>


extern "C" int gethostname(char *name, int namelen);

extern "C" int ftime(struct timeb *tp);

const int NAME_SIZE = 50;

main(int argc, char **argv)
{
    int datasock;
    struct hostent *hp;
    char thisHost[NAME_SIZE];
    struct in_addr this_address;
    double time1,   //start time in seconds using decimals for milli seconds
           time2;   //end time in seconds using decimals for milli seconds


    union sock
    {
       struct sockaddr s;
       struct sockaddr_in i;
    }sock;

    if (argc != 3) {
      cerr<<"YOU NEED 2 arguments:  side2 file port"<<endl;
      return 0;
    }
```

```cpp
int portnum = atoi(argv[2]);

gethostname(thisHost,NAME_SIZE);
hp = gethostbyname(thisHost);
strncpy((char*)&this_address,hp->h_addr_list[0], sizeof(in_addr));
sock.i.sin_port = htons(portnum);
sock.i.sin_addr = this_address;

datasock = socket(AF_INET, SOCK_STREAM, 0);

int fileid;
struct stat fileStat;
stat(argv[1],&fileStat);
char *file_buf = new char[fileStat.st_size+1];

fileid = open(argv[1],O_RDONLY);

//read the initial file into a buffer
read(fileid,file_buf,fileStat.st_size);
close(fileid);

//bind to a port given at the command line
if (bind(datasock, &sock.s, sizeof(sockaddr)) < 0) {
   perror("in the bind");
   return(-11);
}

//now do the listen, waiting for side1 to connect
if (listen(datasock,1) < 0) {
   perror("in the listen");
   return -11;
}
else{
  cerr<<"Doing a listen on socket: "<<datasock<<" for host "<<thisHost<<endl;
}

int addr_len = sizeof(sockaddr);

//side1 connects
int test = accept(datasock,&sock.s,&addr_len);

if (test < 0) {
```

```cpp
            perror("in the accept");
            return -1;
    }

    int len = 0;
    int total = 0, counter = 1;
    timeb *startTime = new timeb;
    timeb *stopTime = new timeb;

    char *temp_buf = new char[10000];

    //now loop for a while, waiting for user interruption
    while (total == 0) {

        ftime(startTime); //start the timer

        //read the file from side1
        while ((len = read(test,temp_buf,10000) ) > 0) {
            total += len;

            if (total == fileStat.st_size) {
                break;
            }
        }

        //send the file to side1
        if ((len = write(test,file_buf,fileStat.st_size )) < 0) {
            perror("Error in write");
            return -1;
        }
        else {
ftime(stopTime);  //round trip over
            time1 = startTime->time + (startTime->millitm/1000.0);
            time2 = stopTime->time + (stopTime->millitm/1000.0);
            cout<<"Round trip #"<<counter<<"time was: "<<time2-time1<<endl;
        }
        counter++;
        total=0;
    }
    close(test);
    return 1;
}
```

APPENDIX E. C++ CODE FOR COMM SERVER STATISTIC REPORTING

```
//------------------------commserver.C-----------
//This is to test a binding to the CommSrv
//and output QoS stats
//----------------------------------------------

#include "comsrv.hh"
#include <iostream.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/timeb.h>

extern "C" int ftime(struct timeb *tp);

main ()
{

    CS_CommServer_var cs;    //pointer to Comm Server
    CS_Endpoint       here, there; //endpoints
    CS_Flowspec       flow;  //flow spec structure
    CS_QOS            nqos; //holds the QoS


    here.CSE_ipaddr = 0x80318203;  //jtfweb3
    here.CSE_id = 1;
    there.CSE_ipaddr = 0x80318204; //jtfweb4
    there.CSE_id = 1;

    //Found that this is not currently used in the Comm Server
    flow.CSF_type = CS_sp_singlexfer; // type of service path
    flow.CSF_dataRate = 6000000; // bandwidth of data flow (bps)
    flow.CSF_packetLength = 8192; // Maximum length of packets (bytes)
    flow.CSF_totalData = 75000; // Total data to be transferred (Kb)

    timeb *startTime = new timeb;

    //bind to Comm Server
    try {
```

```cpp
      cs = CS_CommServer::_bind (":CommServer","");
   }
   catch (CORBA::SystemException& se) {
      cerr <<"Bind to CommServer failed: ";
      cerr <<"unexpected exception" << endl <<se.id() <<endl;
      exit(-1);
   }

   cout<<"Got commserver: "<<endl <<cs->_object_to_string() <<endl;

   //loop for as many time as needed for, each loop is about 10seconds
   for (int ix = 0; ix < 90; ix++) {

      //get the QoS from the Comm Server
      try {
         nqos = cs->CS_CommServer_RequestQoS (here, there, flow);
      }
      catch (CS_XQos_Pred_Not_Avail& ex) {
         cerr << "Get a QoS failed" << endl << ex.id()<<endl;
         exit (-1);
      }
      catch (CORBA::SystemException& ex) {
         cerr << "Get a QoS failed" << endl << ex.id()<<endl;
         exit (-1);
      }

      ftime(startTime);
      cout<<"\nRESULTS FROM Quality Of Service at time: "
          <<startTime->time<<"\n"<<endl;
      cout<<"This is the Bandwidth Range:\n"
          <<"       Low Val: "<<nqos.CSQ_bwLow<<"\n"
          <<"       Hi Val:  "<<nqos.CSQ_bwHi<<endl;
      cout<<"This is the Delay in ms:\n"
          <<"       Low Val: "<<nqos.CSQ_delayLow<<"\n"
          <<"       Hi Val:  "<<nqos.CSQ_delayHi<<endl;
      cout<<"This is the Error Rate:\n"
          <<"       Low Val: "<<nqos.CSQ_errRateLow<<"\n"
          <<"       Hi Val:  "<<nqos.CSQ_errRateHi<<endl;
      cout<<"This is the Latency in ms:\n"
          <<"       Mean:    "<<nqos.CSQ_meanLatency<<endl;
      cout<<"This is the maximum Latency in ms:\n"
          <<"       Max:     "<<nqos.CSQ_maxLatency<<endl<<endl<<endl;
```

```
        sleep(10); //output next set of stats in 10 seconds
    }

    return 0;
}
```
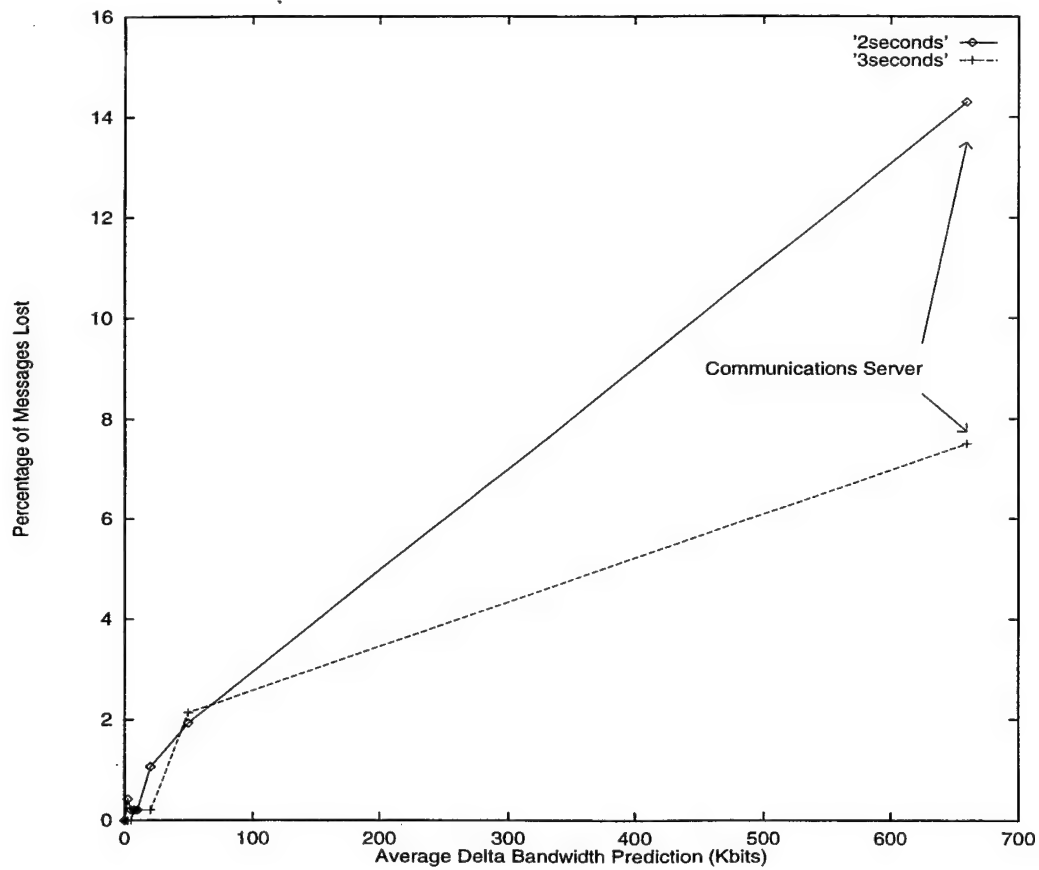
APPENDIX F. ADDITIONAL SIMULATION RESULTS

Figure 35. Percentage of adaptive messages not received by deadline when using Strategy 1 and 100% of messages are adaptive.
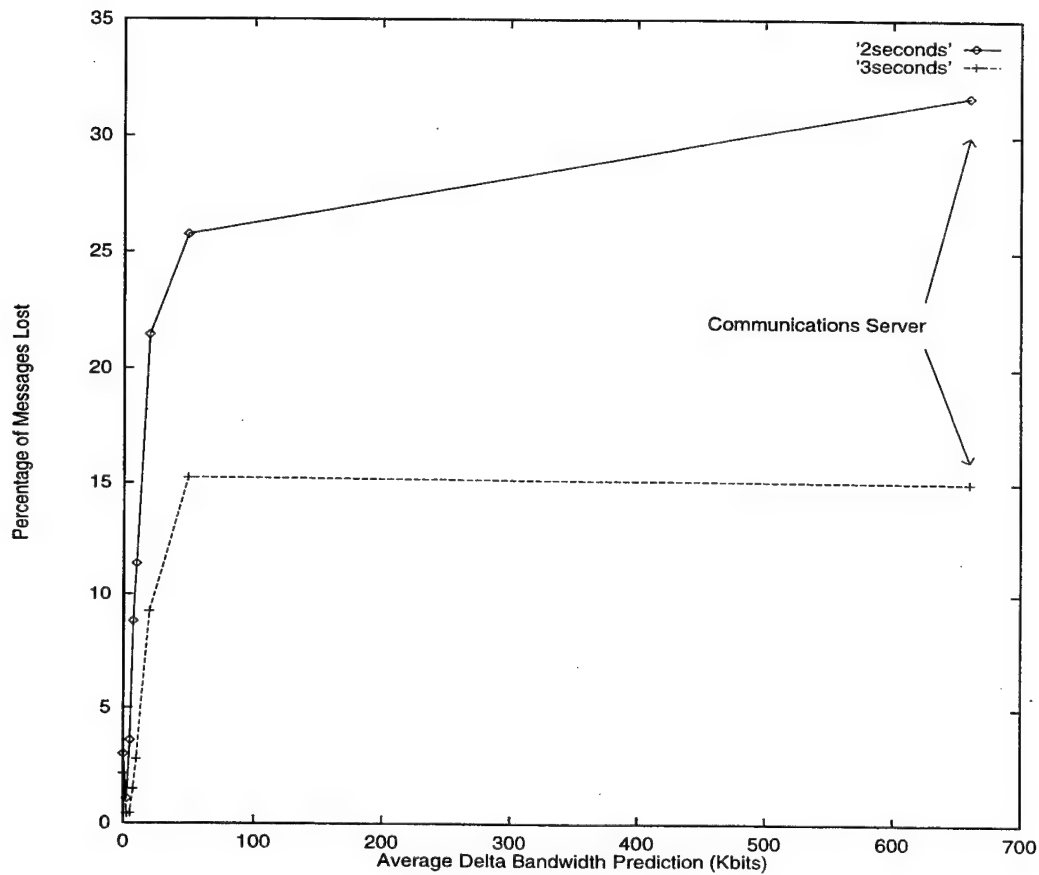
Figure 36. Percentage of adaptive messages not received by deadline when using Strategy 1 and 1.25% of messages are adaptive.
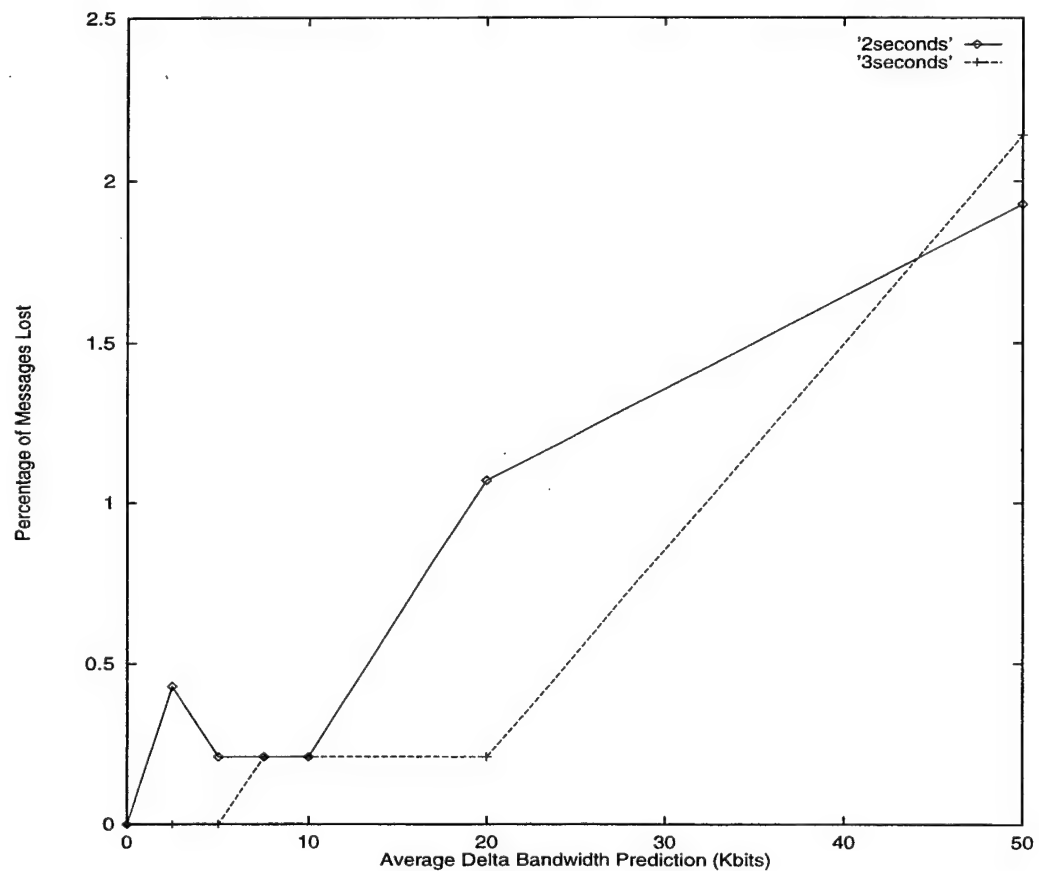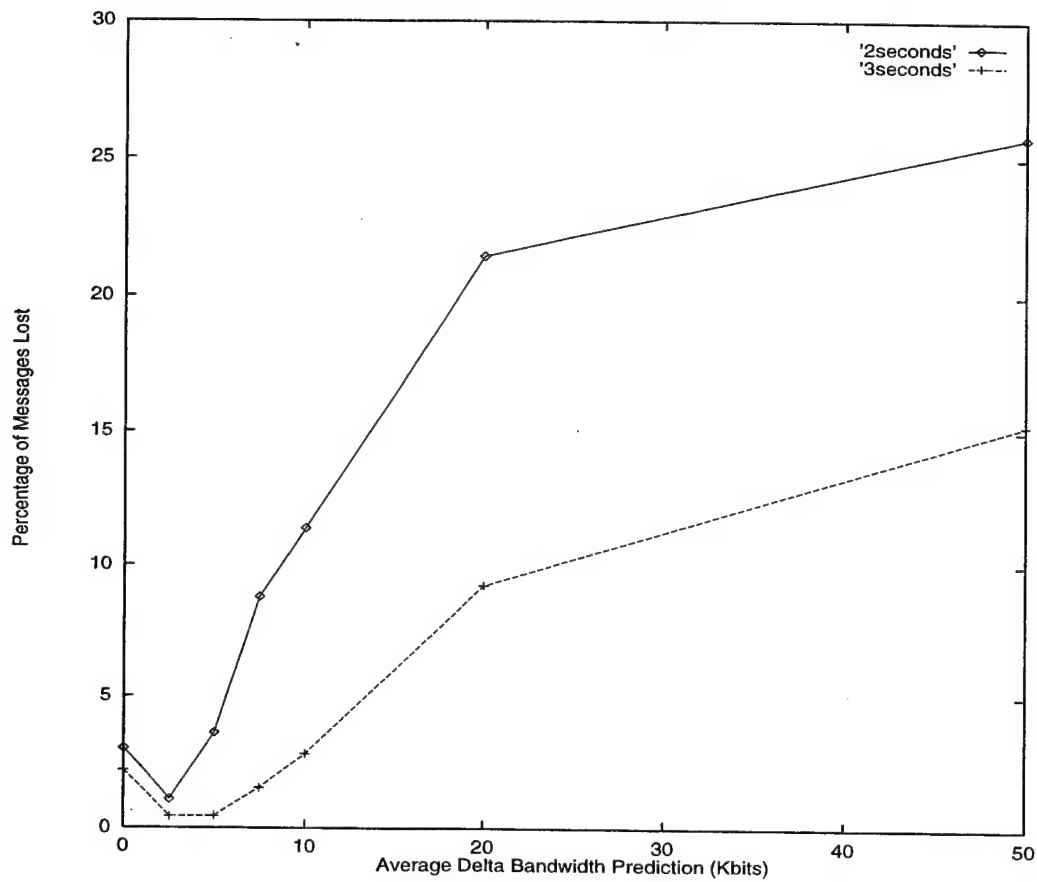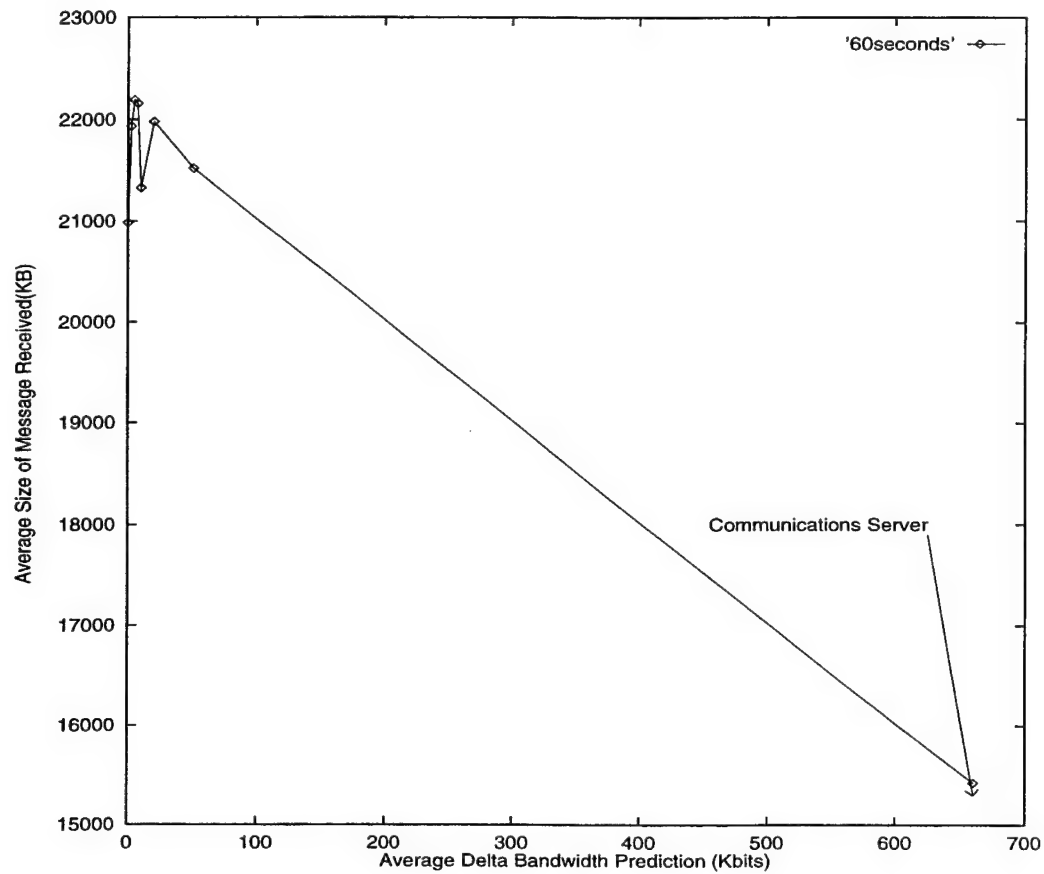
Figure 37. Percentage of adaptive messages not received by deadline when using Strategy 1 and 100% of messages are adaptive.

Figure 38. Percentage of adaptive messages not received by deadline when using Strategy 1, and 1.25% of messages are adaptive.
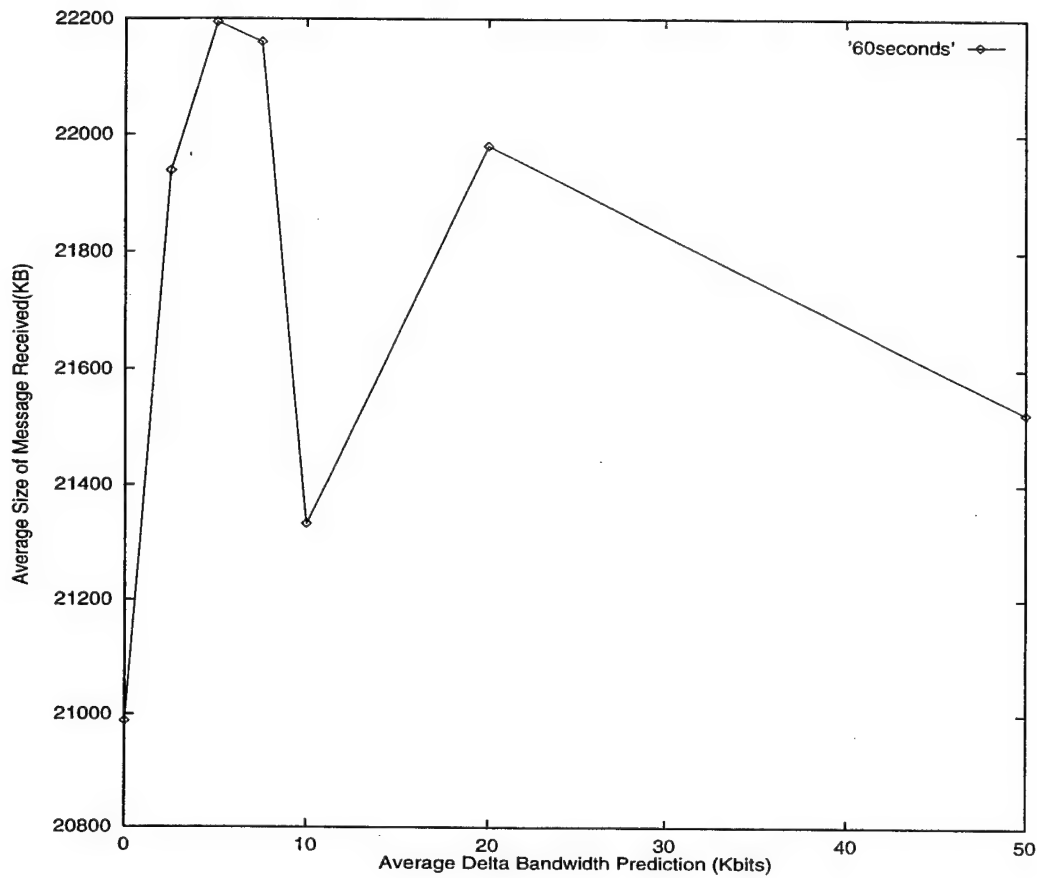
Figure 39. Average size of successful adaptive messages using Strategy 1 when 100% of the messages are adaptive and the mean interarrival time is 60 seconds.
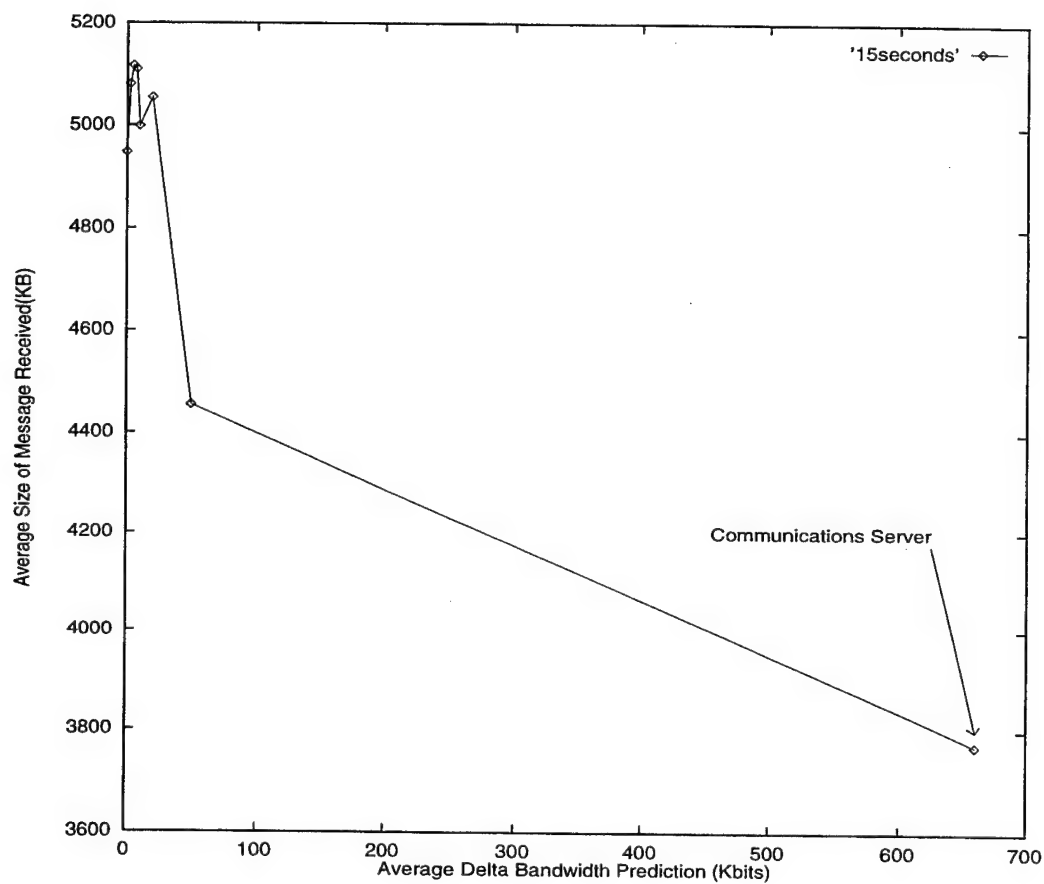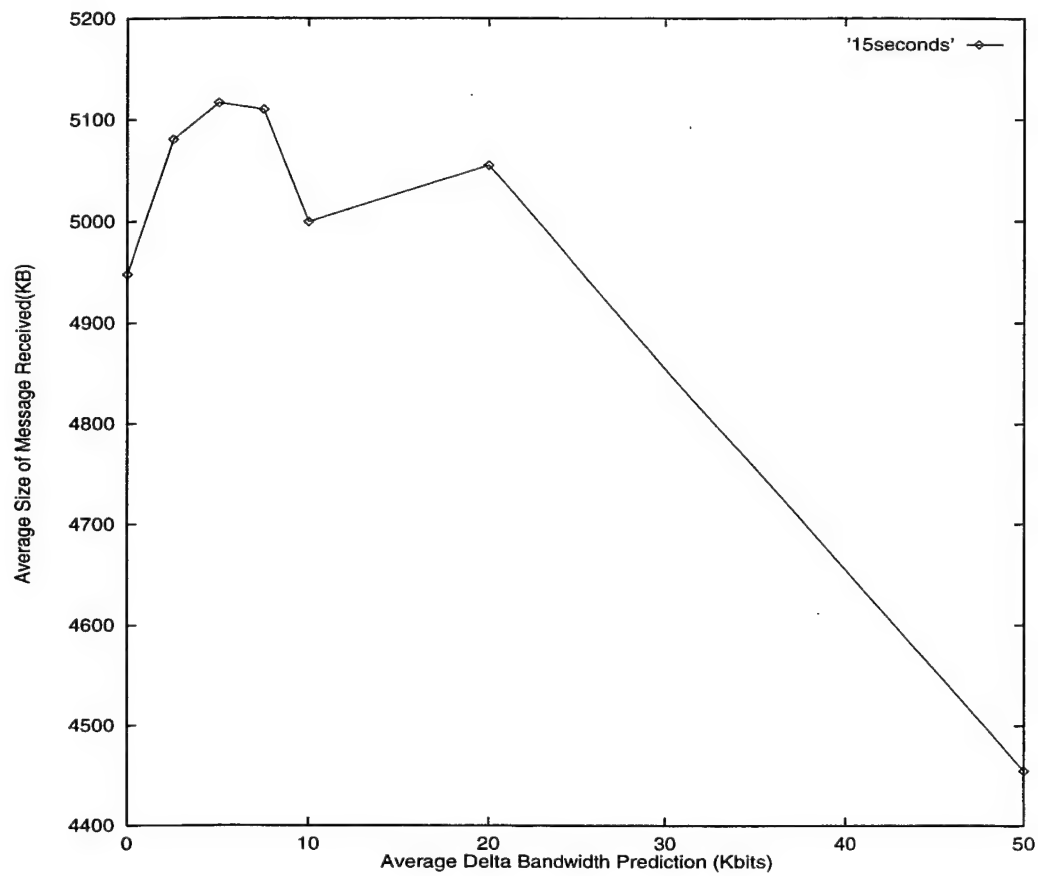
Figure 40. Average size of successful adaptive messages using Strategy 1 when 100% of the messages are adaptive and the mean interarrival time is 15 seconds.
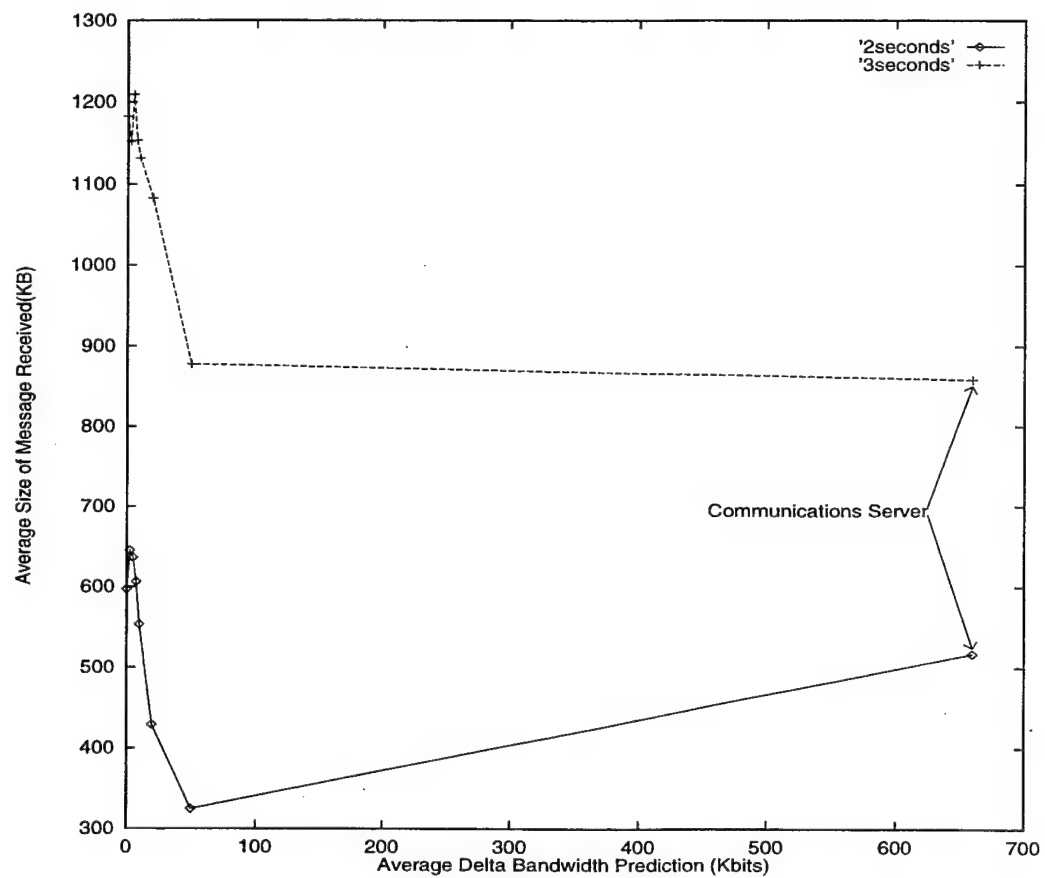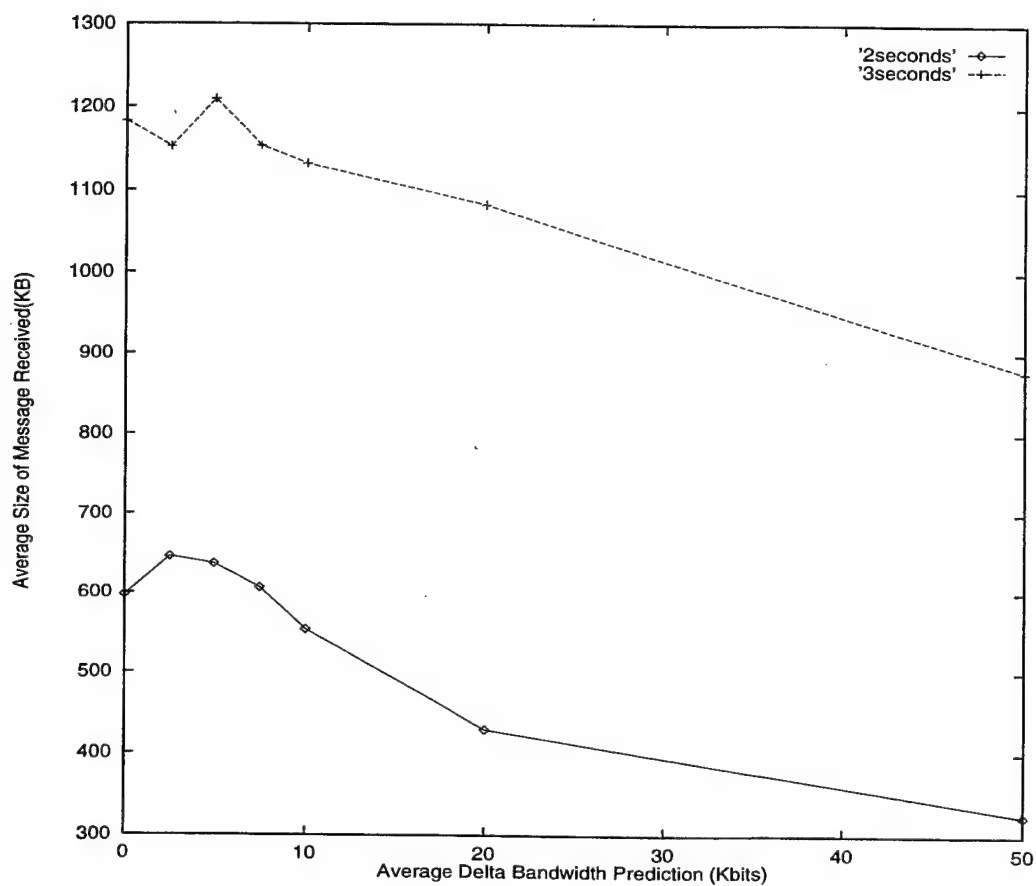
Figure 41. Average size of successful adaptive messages using Strategy 1 when 100% of the messages are adaptive and the mean interarrival times are 2 and 3 seconds.

Figure 42. Average size of successful adaptive messages using Strategy 1 when 100% of the messages are adaptive and the mean interarrival time is 60 seconds.

Figure 43. Average size of successful adaptive messages using Strategy 1 when 100% of the messages are adaptive and the mean interarrival time is 15 seconds.

Figure 44. Average size of successful adaptive messages using Strategy 1 when 100% of the messages are adaptive and the mean interarrival times are 2 and 3 seconds.
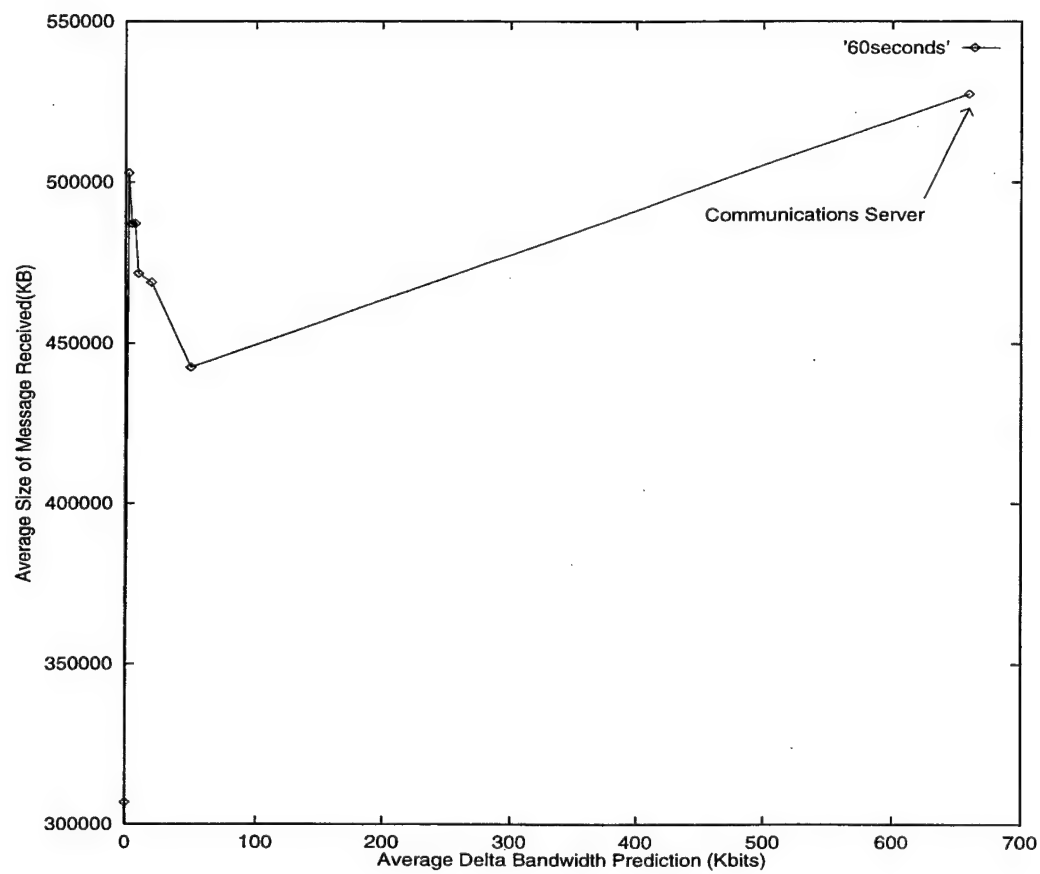
Figure 45. Average size of successful adaptive messages using Strategy 1 when 1.25% of the messages are adaptive and the mean interarrival time is 60 seconds.
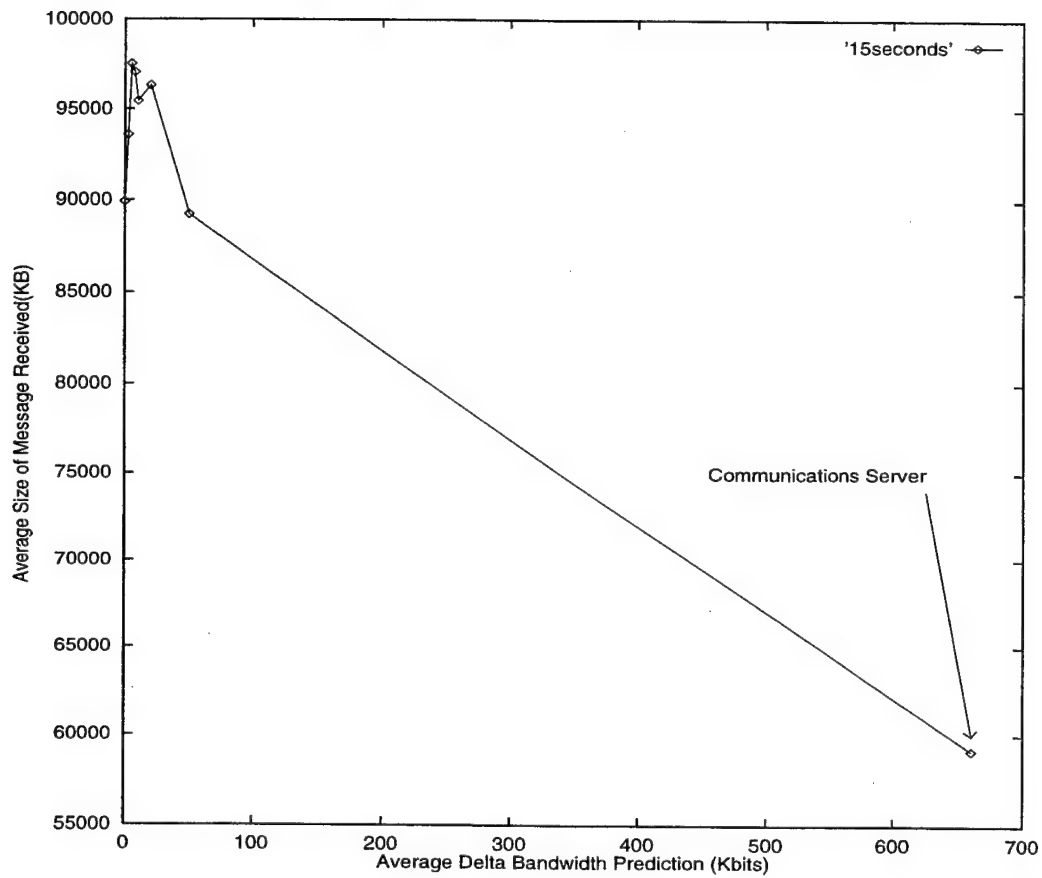
Figure 46. Average size of successful adaptive messages using Strategy 1 when 1.25% of the messages are adaptive and the mean interarrival time is 15 seconds.
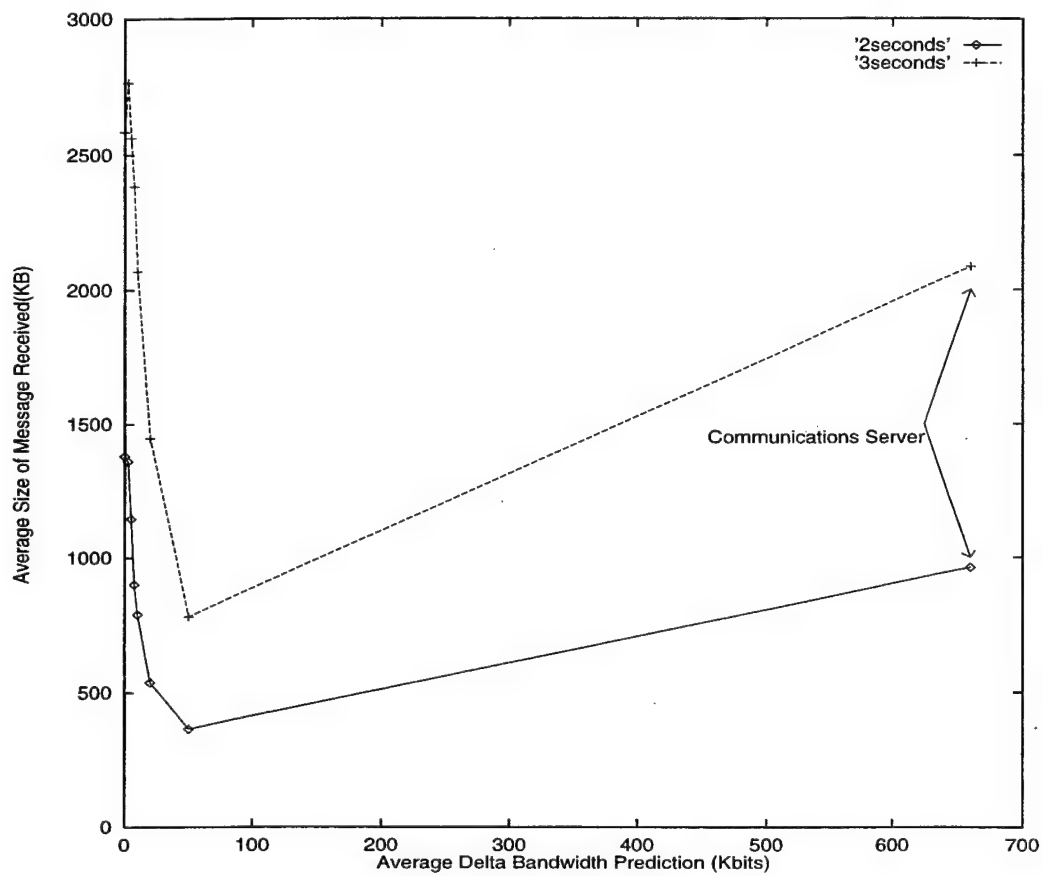
Figure 47. Average size of successful adaptive messages using Strategy 1 when 1.25% of the messages are adaptive and the mean interarrival times are 2 and 3 seconds.

Figure 48. Average size of successful adaptive messages using Strategy 1 when 1.25% of the messages are adaptive and the mean interarrival time is 15 seconds.

134

Figure 49. Average size of successful adaptive messages using Strategy 1 when 1.25% of the messages are adaptive and the mean interarrival times are 2 and 3 seconds.
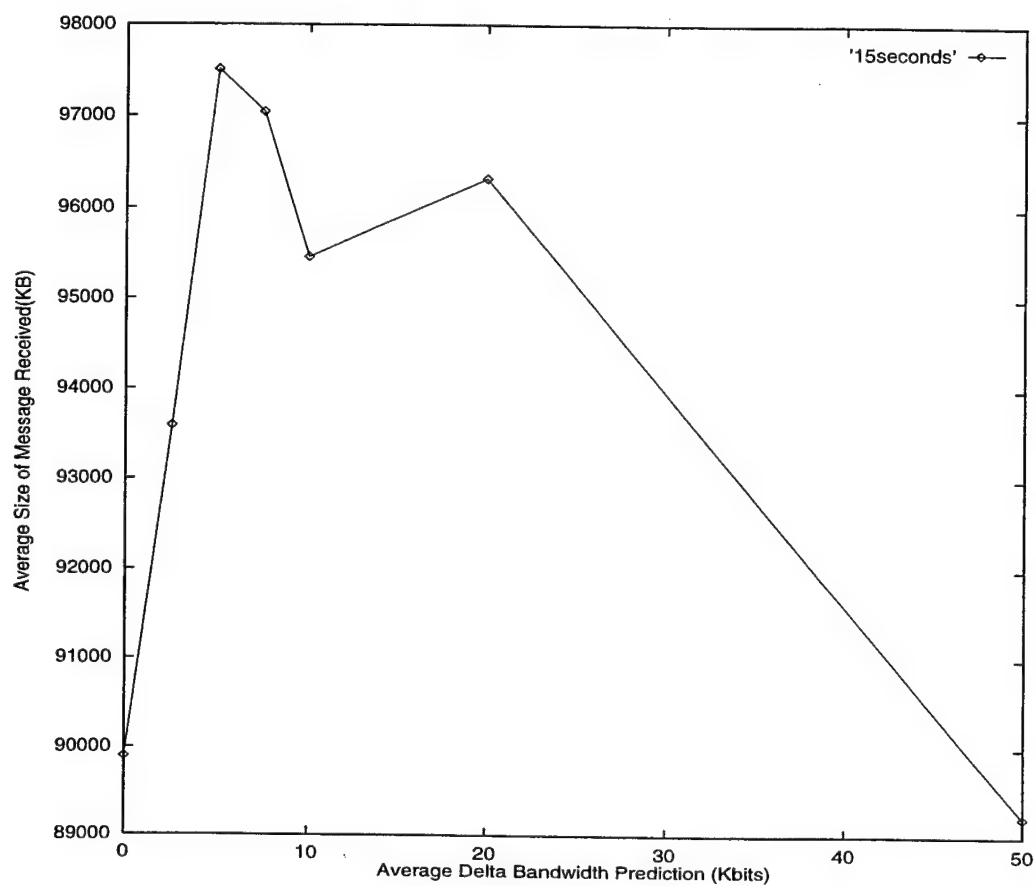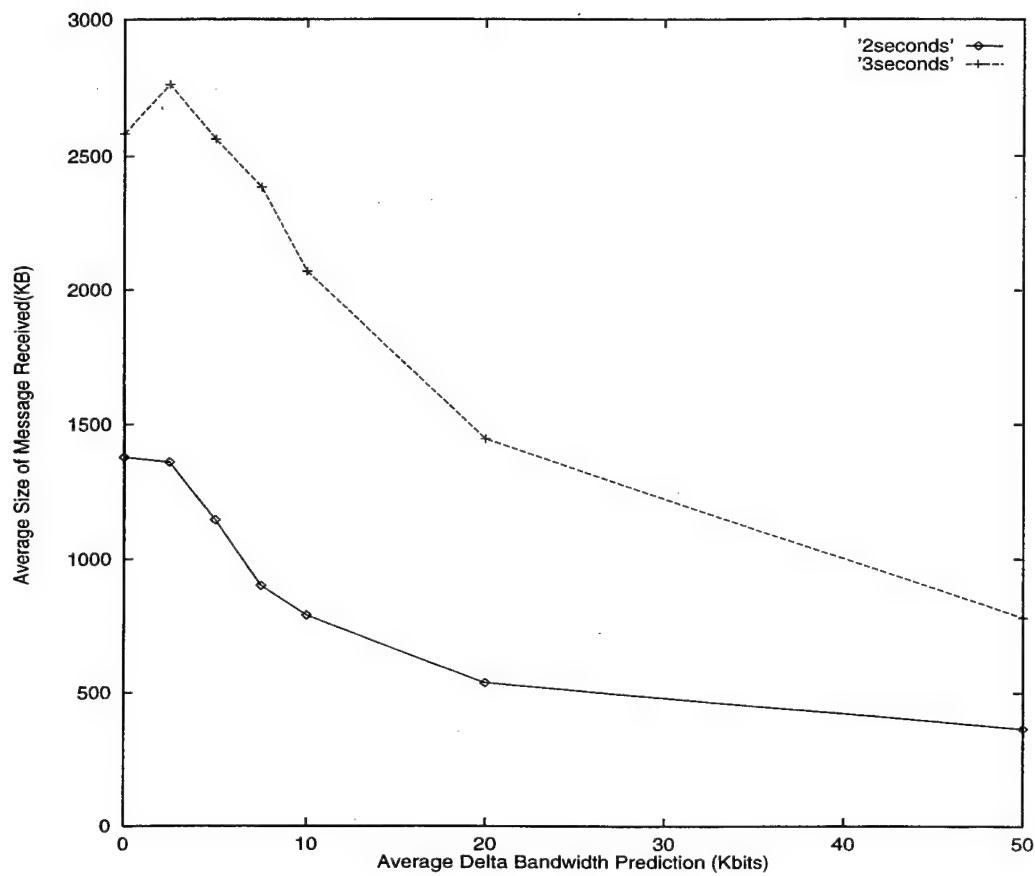
135

Figure 50. Average size of successful adaptive messages using Strategy 2 when 100% of the messages are adaptive and the mean interarrival time is 60 seconds.

Figure 51. Average size of successful adaptive messages using Strategy 2 when 100% of the messages are adaptive and the mean interarrival times are 2 and 3 seconds.
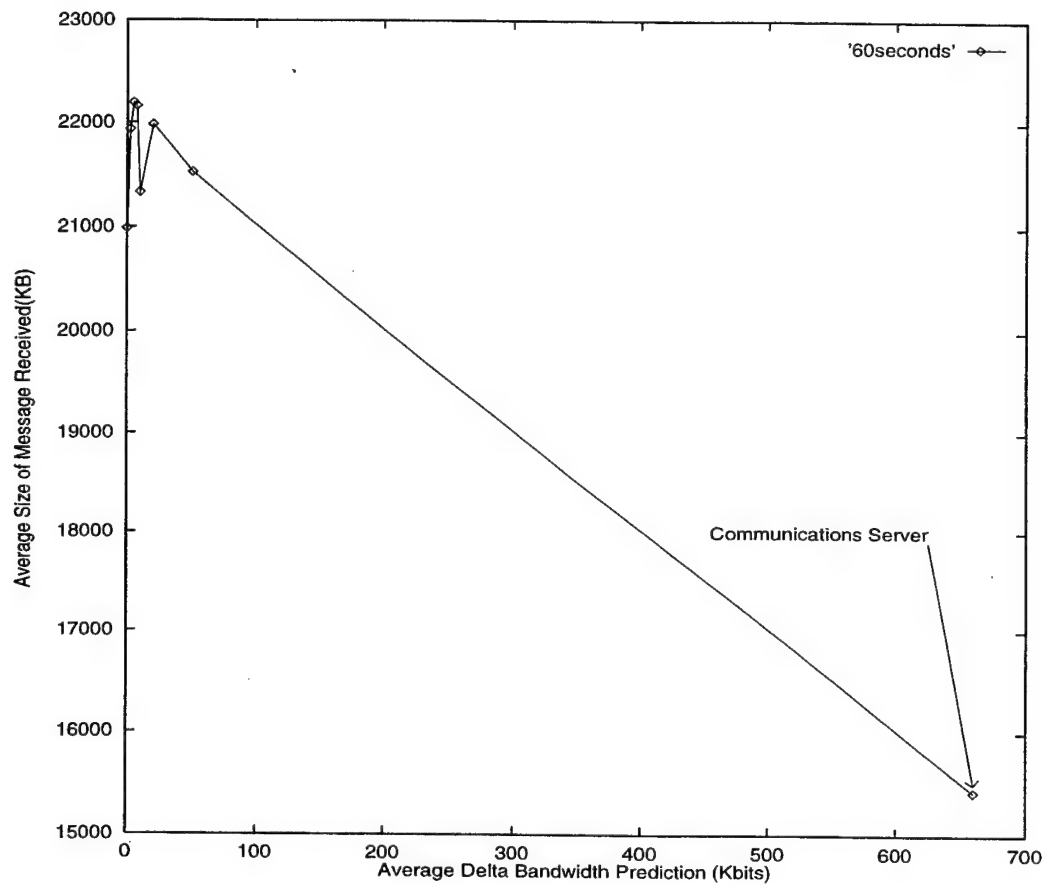
Figure 52. Average size of successful adaptive messages using Strategy 2 when 100% of the messages are adaptive and the mean interarrival time is 60 seconds.

Figure 53. Average size of successful adaptive messages using Strategy 2 when 100% of the messages are adaptive and the mean interarrival time is 15 seconds.
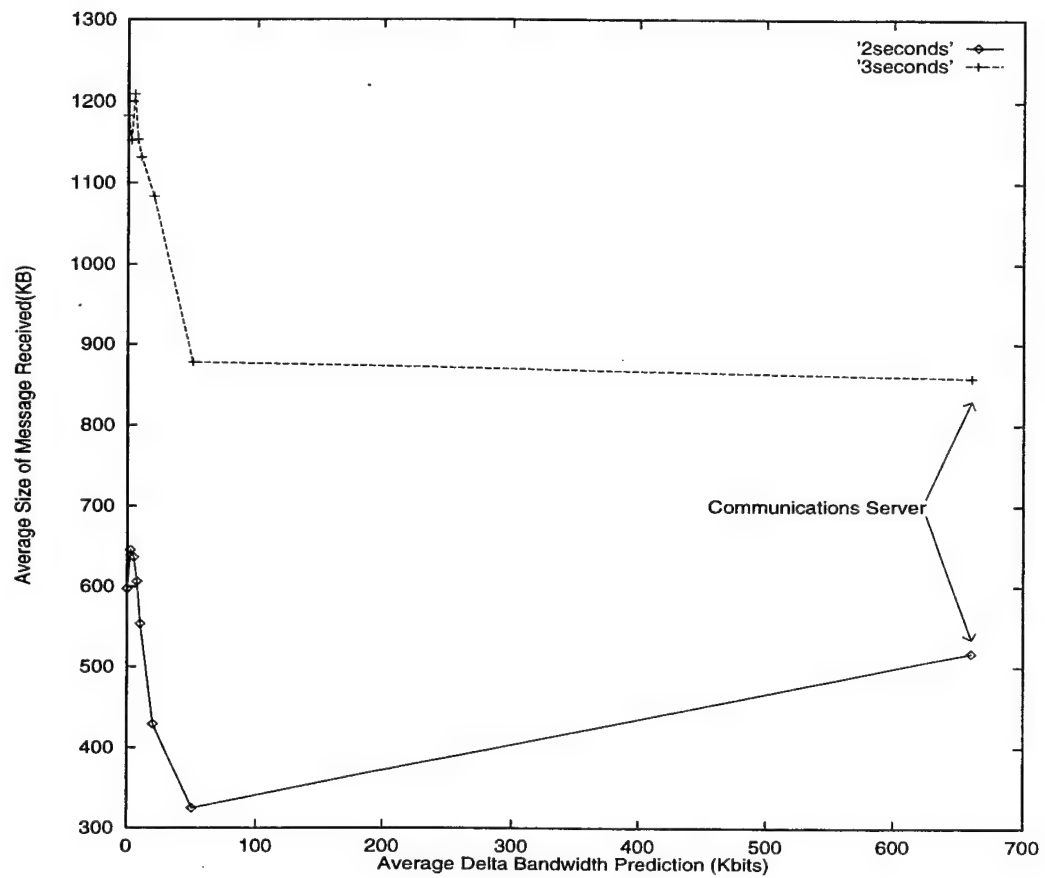
Figure 54. Average size of successful adaptive messages using Strategy 2 when 1.25% of the messages are adaptive and the mean interarrival time is 15 seconds.

140

Figure 55. Average size of successful adaptive messages using Strategy 2 when 1.25% of the messages are adaptive and the mean interarrival times are 2 and 3 seconds.
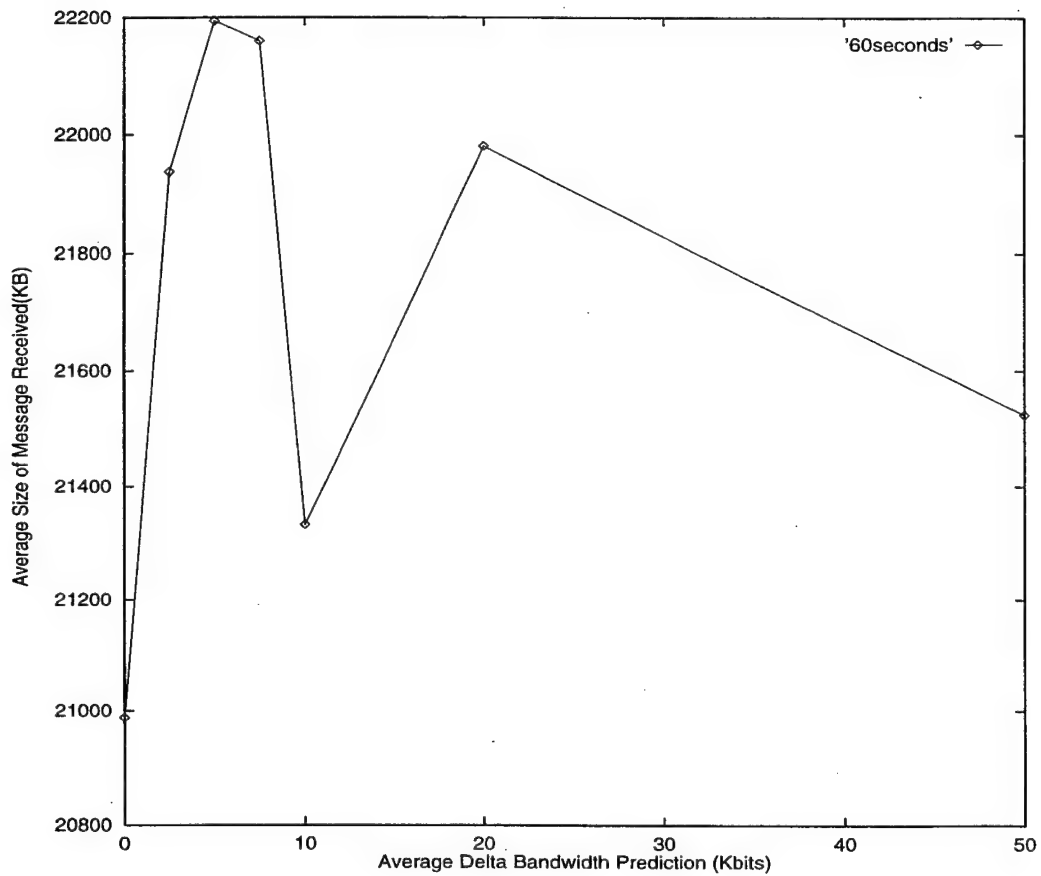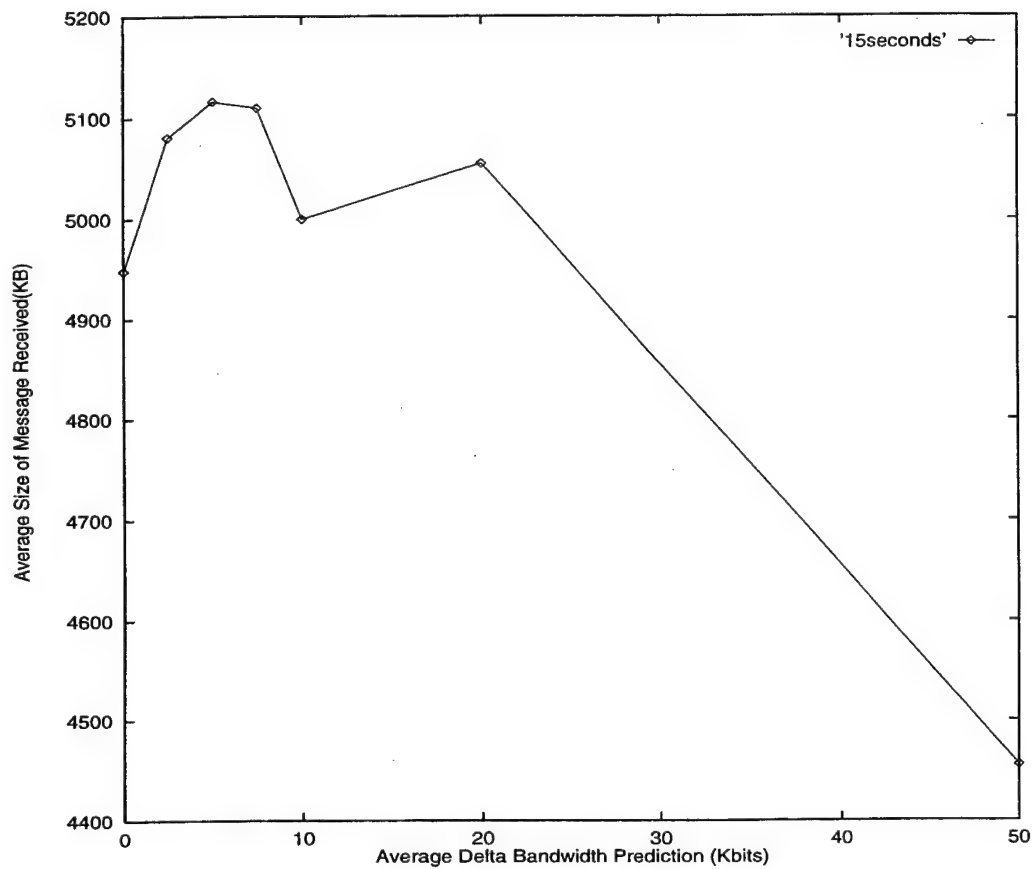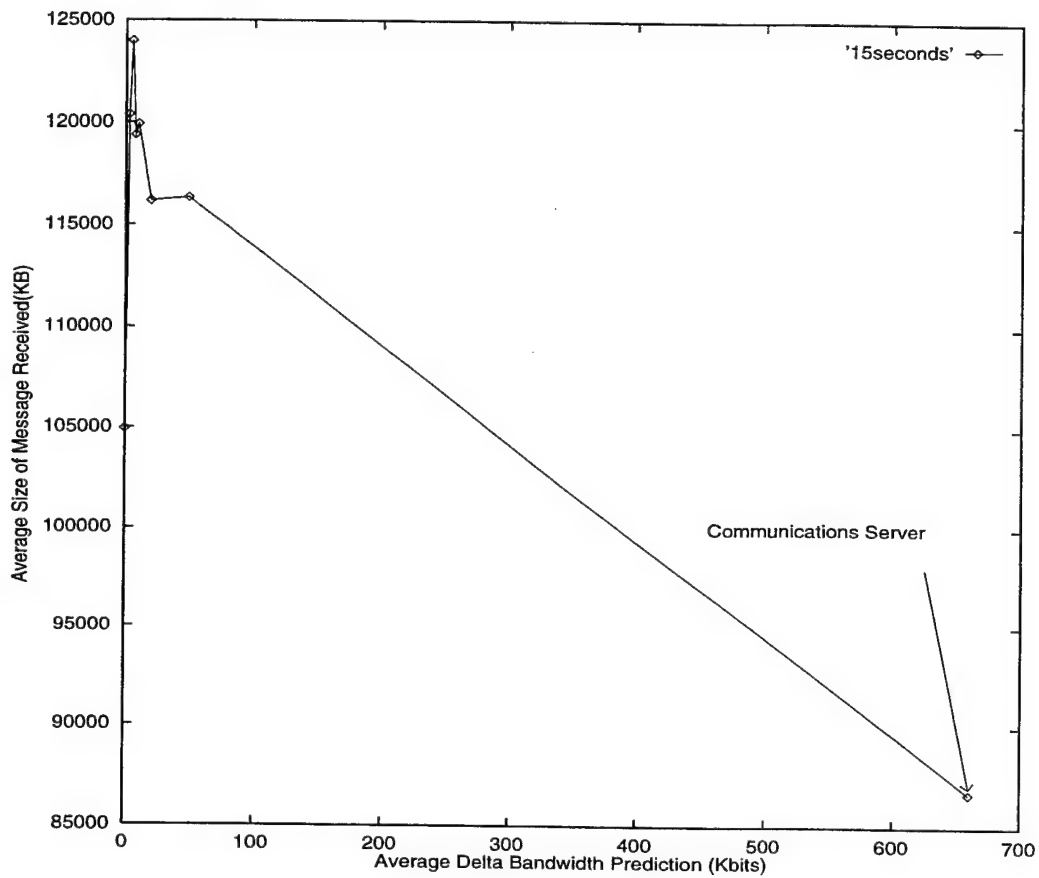
Figure 56. Average size of successful adaptive messages using Strategy 2 when 1.25% of the messages are adaptive and the mean interarrival time is 15 seconds.
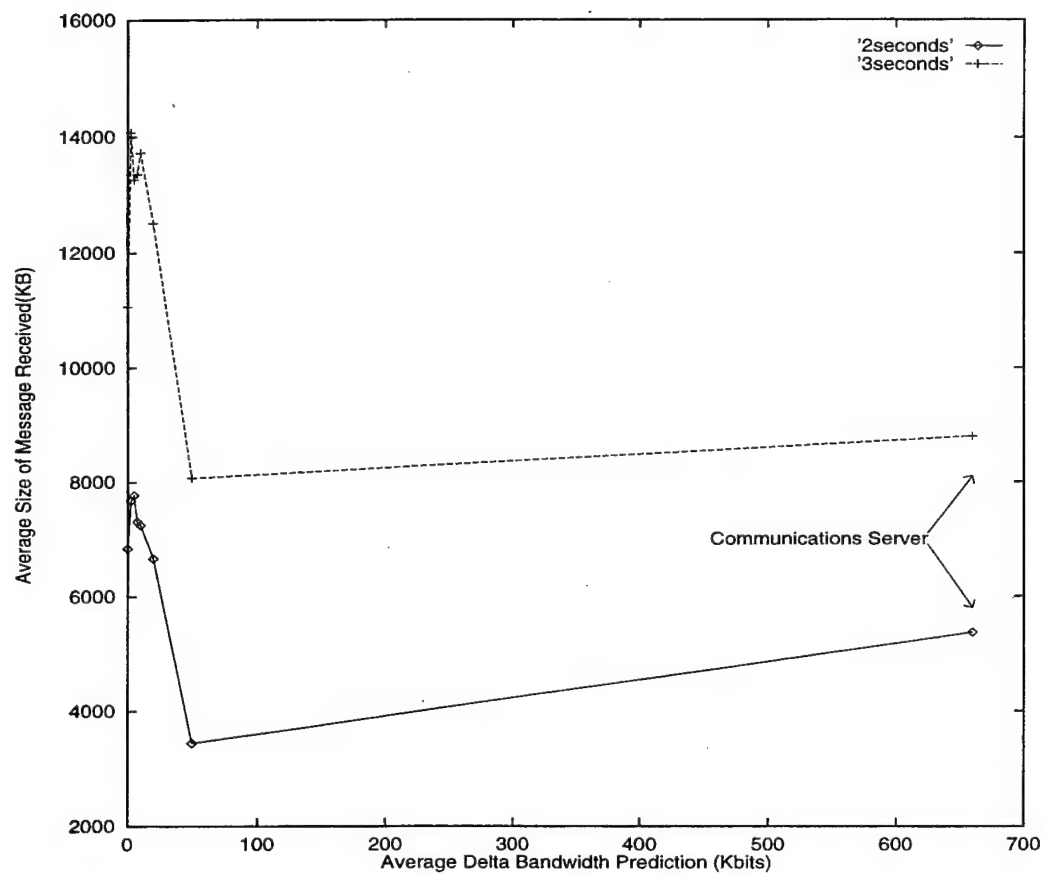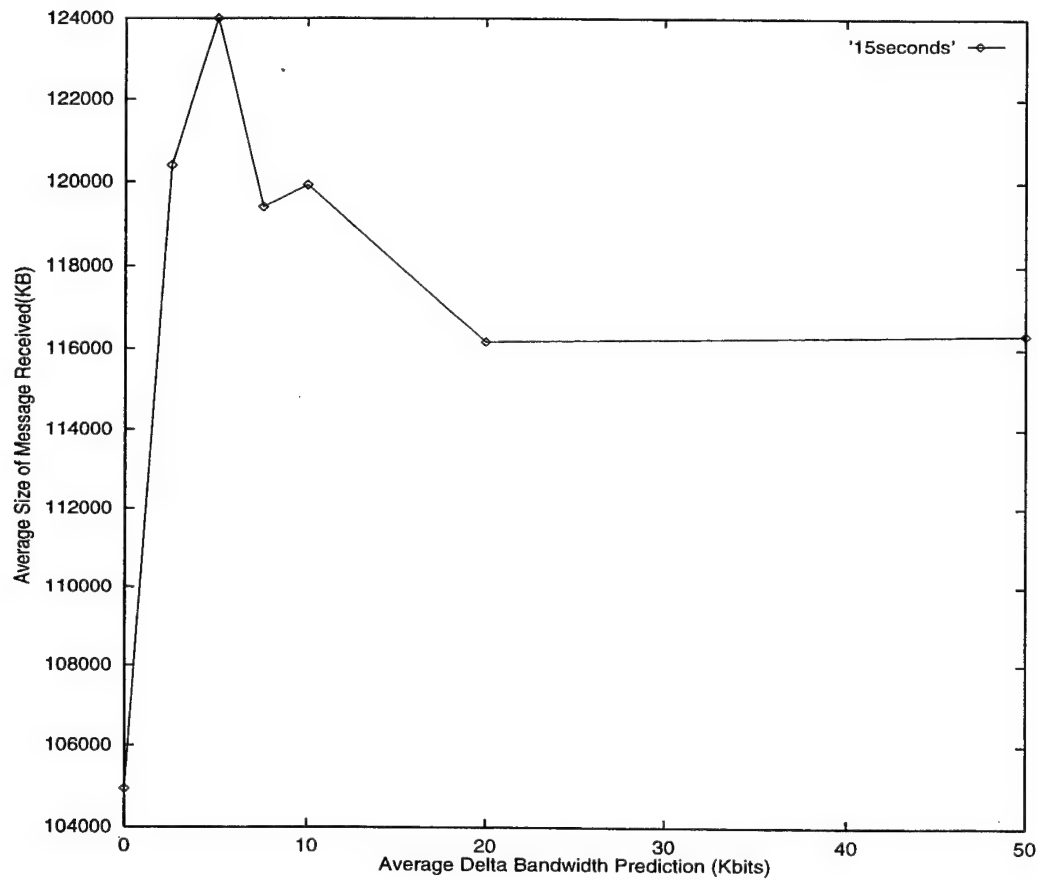
APPENDIX G. LIST OF SYMBOLS AND FUNCTIONS

The following is a list of symbols and functions that were presented in Chapter VI.

$Application_j$ An application that is adaptive and can receive/send data in different formats.

D A collection of data to be received/sent by an adaptive application.

$T_{D,j}$ A specific time after which the data, D, is considered late and is no longer required by $Application_j$.

$F_{m,D,j}$ The different formats of D that $Application_j$ could send/receive. The number of formats, m, depends on the specific D and $Application_j$.

ρ_i Reflects the desirability of $F_{i,D,j}$. For example, an application may accept data in one of two different formats where their normalized priorities are .9 and .1, meaning that the first format is much preferable to the second.

P_j A priority of $Application_j$ that reflects its importance and that of its user, with respect to other applications and their users.

$M_{(i,j,K)}$ A function that indicates the amount of a resource K that was used by $Application_j$ to deliver part of format i.

R_K The acual amount of resource K needed to deliver an entire format.

$U_{K,T}$ The amount of resource K that is available until deadline T.

I_{mode} Indicates what type of environment the system is operating in. For example, **critical** mode could be when resources are under heavy load, as opposed to **normal** mode when there is little competition for resources.

APPENDIX H. RESERVATION PROTOCOLS FOR REAL-TIME DATA

Supporting Real-Time Data on the Internet

by John Kresho

for CS4920, Professor Geoffery Xie

3 Apr 97

1. BACKGROUND

The Joint Task Force Advanced Technology Demonstration (JTF-ATD) Architecture is a suite of software which is being developed by DARPA. This suite of software will aid Joint planners to electronically collaborate on a battle plan during a crisis situation.

This suite of software contains several servers that other applications use to integrate their functions into the JTF-ATD architecture. Some of these servers include:

- <u>Data Server</u> - employs a common object-oriented C2 schema to provide its clients periodically updated query-base views of distributed, heterogeneous databases

- <u>Web Server</u> - provides its clients means to construct, distribute, view, edit,and replicate node-link structures incorporating objects of arbitrary types

- <u>Situation Server</u> - enables its clients to develop interpretations or "pictures" of the battle space incorporating objects, aggregates, inferences, and predictions, all of which are indexed over space, time, and assumed context

- <u>Plan Server</u> - enables a group of distributed planners to hypothesize, evaluate, and disseminate alternative courses of actions (COAs)

- <u>Model Server</u> - sets up and executes simulations to assess alternative plans in the context of various assumed situations

- Map server - constructs and renders "maps", which are selected subwebs (including situations and plans), using appropriate symbologies and geospatial registration

- Communications Server - interacts with client applications to provide information on the current Quality of Service (QoS) on the network

In addition to reporting the QoS on the network, the Communications Server is designed to reserve bandwidth. However currently this portion of the Communications Server is not implemented. With the explosion of live video and audio over the internet, especially with the use of Video Teleconference (VTC), it is vital for the network to provide a guaranteed QoS. If video and audio packets are delayed, or never reach their destination, communication is greatly affected.

This paper discusses the current protocols that are used to ensure timely delivery of real-time data over the internet. It also talks about the next step that must be taken to guarantee a QoS for a particular application. These issues will be the next hurdle for the Communications Server.

2.   BASICS OF ENSURING TIMELY DELIVERY

If an application is the sole user on a given computer, it has the use of all resources on that computer. It does not have to wait for other processes to be serviced. For instance, a VTC application is compressing video to send over the network, it will require the CPU and the network interface. The CPU immediately services these compression requests, and the network interface will immediately service these data packets and put them on the network.

Unfortunately, once these data packets are placed on the network, they must compete with other data currently there. The VTC data packets must wait for their turn to be serviced. The reader can probably relate this to sitting in a traffic jam, awaiting your turn to get off the highway.

A direct connection to each of its destinations would provide a computer with a congestion free link, but it is unrealistic and inefficient. (That would be like a car

146

having its own road wherever it went.) In order to provide real-time data with a high
QoS, protocols must be used to communicate an application's requirements to the
computing resources such as the CPU, network interface and the network bandwidth.
(See Figure 57)

Protocols at the transport layer communicate directly with an application,
helping it to determine its QoS demands and speed up processing of the real-time
data. However, in order to provide guarantee for this QoS, resources must be re-
served (similar to an HOV lane on the highway). This resource reservation task is
usually performed by another set of protocols that operate at the network layer. The
following sections discuss two transport layer protocols that help applications achieve
a preferred QoS, and two network layer protocols that perform resource reservation
for a guaranteed QoS.

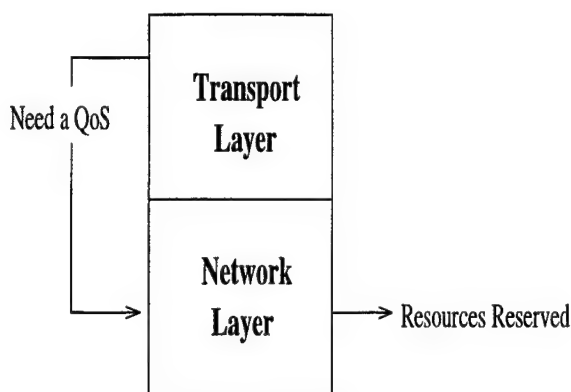Figure 57. Transport and Network Layers

3. REAL-TIME TRANSPORT PROTOCOL (RTP)

The RTP protocol was introduced in January of 1996 by the Audio-Video
Transport Working Group of the Internet Engineering Task Force (IETF) [Ref. 22]. It
is designed to operate at the transport layer (Figure 58). RTP works with applications
to provide an end-to-end delivery service for data that has real-time characteristics.
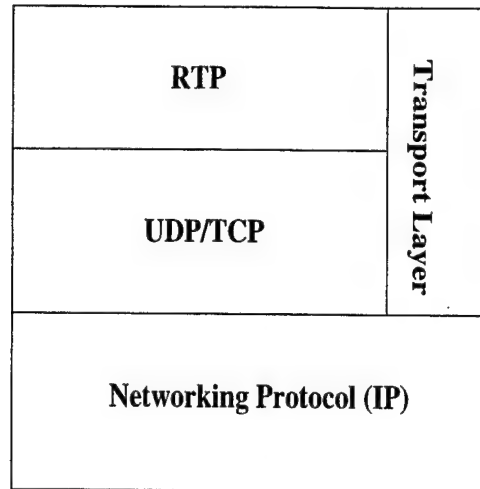
147

Figure 58. RTP in the Transport Layer

RTP interacts with the data produced by applications and provides a method of tracking these data packets at the destination. RTP encapsulates each data packet with several additional fields of information such as:

- Payload Type - format of the audio or video being sent

- Sequence Number - tracks the order in which the packets are produced

- Source ID - who is the originator of the packets

RTP itself does not ensure that packets arrive at the destination in order. In fact, RTP does not ensure that packets will arrive in a timely manner at all, nor does it attempt to reserve resources for its clients. It relies on the underlying network protocols to provide these kind of services. RTP uses its own sequence numbering to allow receivers to reconstruct the ordering of packets, which can speed up processing of video packets greatly because packets do not have to be decoded in order to derive the correct location before viewing.

RTP uses UDP to move its packets. This is due to the fact that UDP does not provide extensive error checking. There is no reason to resend packets which contained errors because real-time data is time sensitive. By the time the new packet is received, it is late and not useful. However, the nature of real-time data, such as

148

video and audio, is that an application can easily interpolate missing data. UDP is not the only protocol that RTP can use to aid in transporting its data packets. It also can use the assistance of the Internet Stream (ST-II) protocol which works at the network layer and will be discussed in Section 5.

Next we discuss another transport protocol for real-time data, the Heidelberg Transport System.

4. HEIDELBERG TRANSPORT SYSTEM (HEITS)

HeiTS is another transport layer protocol that helps to achieve a high QoS for real-time data [Ref. 23]. It is currently designed to use ST-II as its network layer protocol. HeiTS is able to adapt to environments that allow for reservations such as FDDI and ISDN, as well as the ethernet and Token Ring environments that are best-effort environments. This protocol is unique due to its ability to use Media Scaling in order to adapt to the congestion on a network link.

a. Types of Media Scaling

There are two distinct classes of Media Scaling. The first is Non-Transparent Scaling, where HeiTS communicates to the application its needs to adapt to the changing network environment. This usually acts on actual data within a particular stream. The second is Transparent Scaling, where HeiTS acts on the media stream without communicating with the application. Entire streams are usually manipulated in this case.

i. Non-Transparent Scaling

When a network link becomes congested, HeiTS may communicate to the application to tell it to reduce the amount of data being produced. In the case of audio, an application can reduce the sampling rate at which its recording is being done. This will reduce the size of the audio data, reducing the congestion on the network.

When video is involved, applications usually have more options to modify the data as opposed to audio. One common approach is the reduction in frame rate.

149

Other approaches include reducing the number of pixels per image, reducing the number of colors used (i.e. gray scale), or changing the type of encoding technique (i.e. JPEG,MPEG, DVI).

In any case, the data is modified before it gets to the transport layer. HeiTS provides the interface between the application and the network layers in order to control the congestion on the network link.

ii.   Transparent Scaling

Transparent Scaling puts more of a burden on the transport layer protocol. HeiTS does not communicate with the application; it makes its own decisions on when to change the flow of data at the sender and receiver. Transparent Scaling has two stages, Continuous and Discrete Scaling.

Referring to Figure 59, the sender has established several streams between the sender and 2 receivers using the ST-II protocol (which we will discuss in the next section). The basic connection is the baseline QoS (6 frames/sec) which was established initially, but the available resources currently allow for 18 frames/sec. HeiTS setup 2 more separate streams to carry this additional service to the receivers.
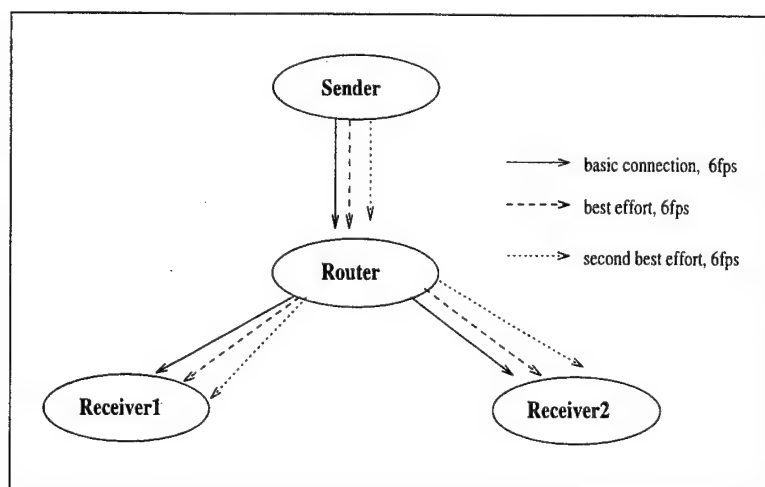

Figure 59. Transparent Media Scaling

During this multicast, the network becomes congested. The receiver first notices this because HeiTS can determine when packets are late by using the max

150

packet delay from the Flow Specification of ST-II. When the number of late packets reaches a certain threshold, the first reaction of HeiTS is to use the Continuous Scaling technique. Since the congestion may be temporary, HeiTS does not immediately cut traffic down to nothing. Packets will be dropped at the receiver's end, and the application will not see these packets. The sender is not effected.

If the congestion persists, the next thing HeiTS will do is to reduce the amount of traffic coming from the sender. This reduction may go all the way to zero, but the stream is still present, along with its reserved resources. At different time intervals, the sender attempts to send more data. If these attempts fail, then the Discrete Scaling technique is used.

Discrete Scaling terminates an entire stream, so it is most effective on video streams. A listener will notice a large discrepancy in service if an entire stream of audio is deleted. In this case, HeiTS will choose the second best effort connection (Figure 59). Now the receivers are getting 12 frames/sec. The same process will continue if the congestion continues, deleting all best effort connections if necessary. However, HeiTS will ensure that the baseline QoS is met, and that connection will not be terminated.

The next two sections discuss protocols which operate at the network layer to reserve resources for the above transport protocols. The first is the ST-II protocol.

5.   INTERNET STREAM PROTOCOL (ST-II)

ST-II was introduced in August of 1995 by the ST-II Working Group of the IETF [Ref. 24]. It operates at the network layer, and Figure 60 shows its position relative to RTP in the protocol stack. ST-II will reserve resources from the originator to the receiver by establishing a stream. Such reservation is necessary for the network to guarantee a certain quality of service(QoS) for clients.
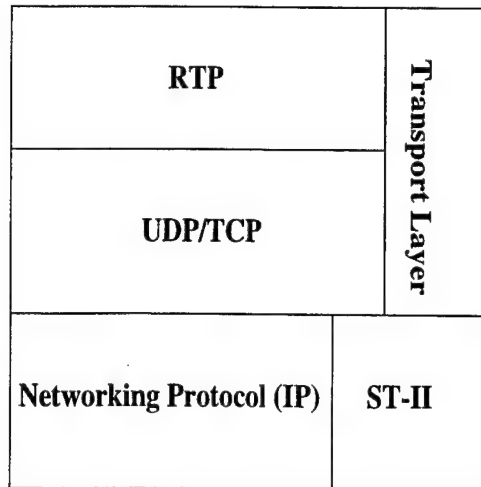
Figure 60. ST-II with Transport Protocols

a. Establishing a Stream

Figure 61 shows an example setup of an originator attempting to set a stream up to two receivers. Initially, the originator asks for a certain QoS that will meet his data requirements. This QoS is contained in a data structure called a Flow Specification. This structure contains such things as the average expected throughput, maximum packet size, and maximum packet delay. In our example, the originator requires that the receivers play video at 12 frames per second, which translates into a maximum packet delay of 20ms from sender to receiver.

This 20ms packet delay, along with other QoS characteristics are placed in the Flow Specification and sent to each receiver. Following the path to receiver 1 (Figure 61), the Flow Specification (FS in diagram) first reaches an intermediate router. This router must decide whether it has the available resources (CPU time, buffer space, bandwidth, etc) to provide the QoS specified in the FS. If it does not, it rejects the request and sends it back to the receiver, who then can request a less demanding QoS.

Let us assume that in this example, the router has the available resources to meet the specified QoS, i.e., the router determines that it can process and forward the packets down the stream in 12ms. It modifies the FS to reflect these changes (i.e. the max packet delay becomes 8ms). The modified FS' is then forwarded down the
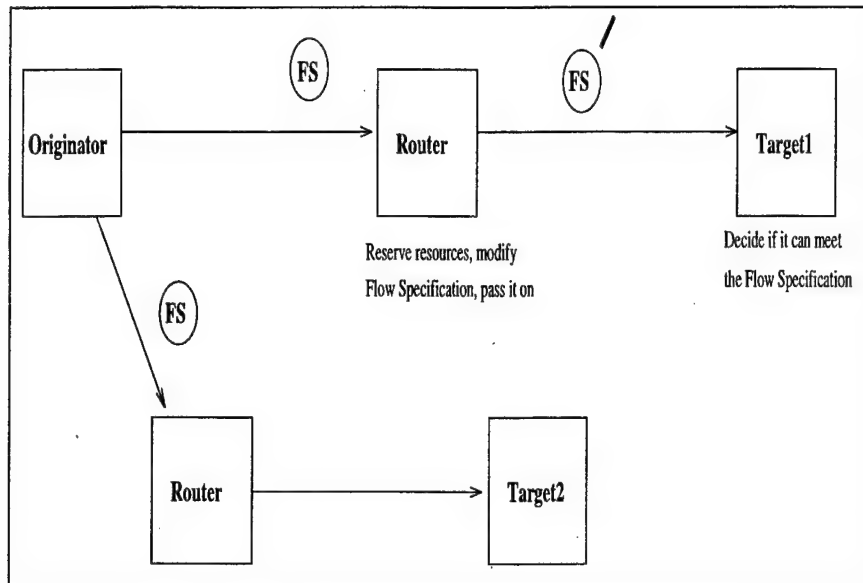
152

Figure 61. Establishing a Stream for ST-II

path towards the receiver. Now the receiver decides if it has the resources to process the packets in 8ms. If not, the request is sent back, freeing up reserved resources and leaving it up to the originator to make another request. Otherwise, the receiver reserves the resources, and the Flow Specification is then propagated back to the originator, letting each intermediate node know that the request is accepted and to set aside the appropriate resources.

b.   Adding Participants to Existing Group

Suppose a multicast is already setup using ST-II and the streams are configured as in Figure 62. The originator is sending 12 frames/sec of video to the nodes on the left hand side of the tree. Suppose that another node wishes to join the multicast, but can only process 4 frames/sec. This node is shown as a dotted circle in Figure 62 . Since ST-II initiates the stream from the originator, another stream must be established to the new node. If several other nodes wish to join below this node, the path's resources can be quickly exhausted.

The originator initiated stream establishment puts a limit on the number of participants that can join the mutlicast due to its excessive use of resources. In a
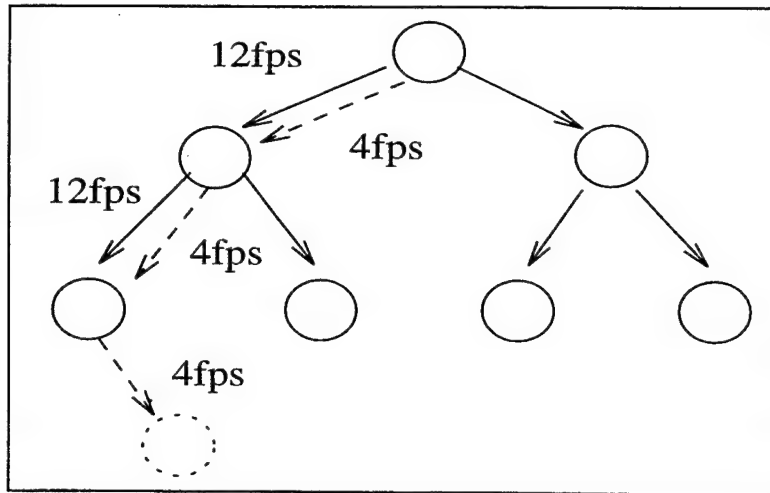
153

Figure 62. Adding a participant using ST-II

military environment, resources will be in high demand. Therefore better resource management is a must. One such protocol that provides this criteria is the Resource Reservation Protocol.

6. RESOURCE RESERVATION PROTOCOL (RSVP)

The latest information on RSPV was published as an Internet Draft in Nov 96 [Ref. 25]. RSVP works at the same layer as ST-II does, the network layer. RSVP is very similar to ST-II, in the fact that it reserves resources along a path and creates a stream between the receiver and originator. However, the method of establishing the streams in RSVP is more efficient in its use of resources.

a. RSVP Stream Establishment

A source application using RSVP begins participation in a group by sending a *Path* message to a receiver (Figure 63). This Path message does two things citeMESZ94:

1. Distributes the flow specification to the receivers

2. Establishes a Path state in the intermediate nodes on its way to the receiver
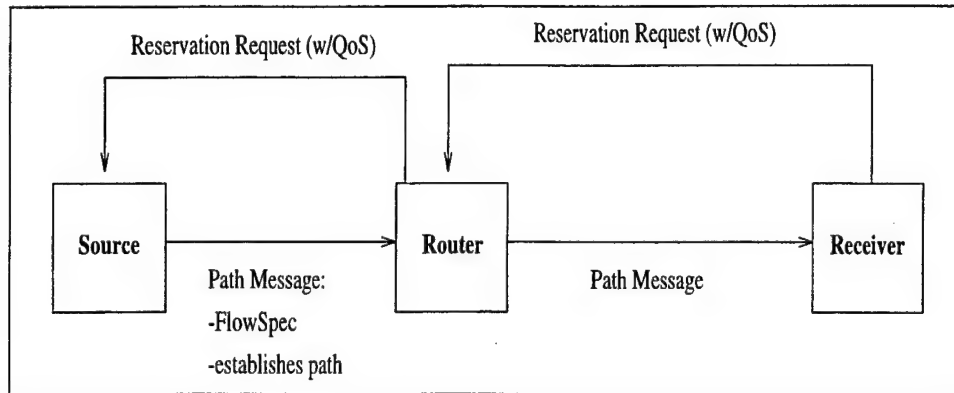
154

Figure 63. RSVP Stream Establishment

Note that no resource reservations have been established yet. Once a receiver obtains the Path message, it must determine what its desired QoS will be, based on its knowledge of its local state and information from the Path message. The receiver then initiates a reservation request back toward the sender, using the Path that was established previously. Each intermediate node reserves the required resources similar to the ST-II method, then passes on the request. This message propagation stops either when it reaches the sender, or when it encounters a node that already is participating in the same group. Once this stream is established, data will begin to flow with the requested QoS.

b. Adding a Participant using RSVP

The receiver-initiated reservation allows RSVP to accommodate heterogeneous receivers' needs. This is where RSVP has the advantage over ST-II and why it will most likely be the protocol of choice in the near future. RSVP adapts to the changing needs of the network by using *soft states*. It allows intermediate nodes to dynamically adapt to the addition or deletion of participants, efficiently using the available resources. The following example will further discuss this method.

Using a setup similar to the ST-II example, Figure 64 shows the sender giving a node 12 frames/sec of video. Then another node wishes to join the multicast group, but it can only process video at 4 frames/sec.
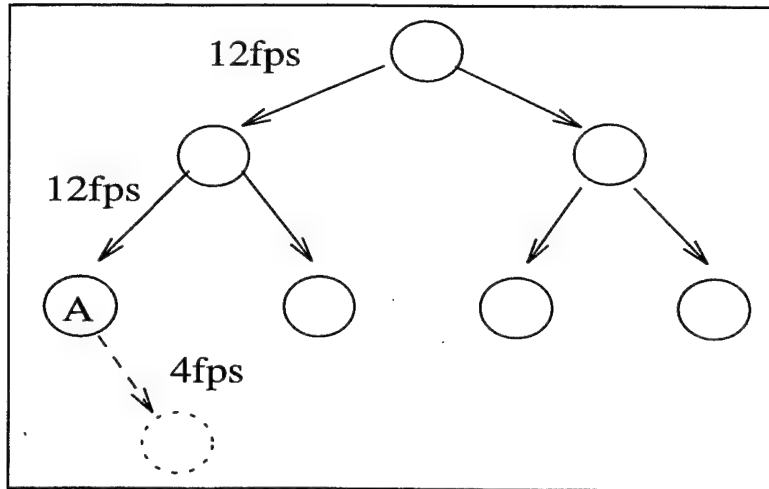
155

Figure 64. Adding a participant using RSVP

When this new node begins its reservation request back towards the sender, it will stop at the first node (A) in its path. Using the *soft state* mechanism, this node will dynamically adapt to this reservation request and begin to send 4 frames/sec of video to the new node. The only resources that were reserved are between the new node and node A. This is much more efficient than ST-II's procedure, and will most likely give RSVP the edge in the future.

7. DISCUSSION

Discussed above are several protocols that serve as interfaces for clients to negotiate a QoS. However, these protocols cannot guarantee a given QoS alone. We must also consider the packet scheduling technique used at the routers between the source and destination. If a router is using a FIFO queue to schedule packets, then real-time data packets will not have any guaranteed performance on delays. For instance, an RSVP agent receives RSVP data packets and hands them off to this FIFO packet scheduler. It is possible that these packets with time constraints will be placed behind a large number of packets which arrived first, but have no time constraints (i.e. FTP packets).

The next step in guaranteeing a given QoS is to develop algorithms that will

156

schedule real-time data packets in a fair and efficient manner. Currently there is a network protocol that may help in the design of such algorithms, IPV6. It is a new version of the Internet Protocol that contains new data fields such as:

- Priority Field - Can instruct routers about the level of serviced required for a packet (Time sensitive vs. non-time sensitive)

- Flow Label - Used to store information about the link to speed datagram processing

- Routing header - Used to override default algorithms. Can direct packets over a specific network link

Developers can use these fields to produce algorithms that recognize time constraints on real-time data and handle the data packets appropriately.

One such scheduling algorithm was proposed in a 1991 IEEE JSAC paper [Ref. 26]. It defines three different queues that packets would be place into. The highest priority queue is a *Deterministic* queue. Packets that have a strict time constraint (i.e 10ms) are placed here. If this queue is empty, then the *Probabilistic* queue will be serviced. This queue contains packets that belong to application that require only a portion of their data to be on time. For instance, an application may only need 85% of it packets on time. The last queue may be serviced when both of the above queues are empty. It contains packets that have no real-time characteristic, e.g. FTP data packets. By scheduling packets in this manner, real-time packets will receive the priority service required.

8. CONCLUSIONS

When it comes time to implement the reservation portion of the Communications Server for the JTF-ATD architecture, it will have to take in consideration the protocols discussed above for QoS guarantees. As this paper has shown, the protocols are in place, and a client interface will only have to be developed to use these protocols in conjunction with the JTF-ATD suite of software.

157

RTP and HeiTS can handle network degradation with respect to real-time data. RSVP will most likely be the predominate protocol due to its nature of handing resources efficiently network wide. However, guaranteeing a QoS is not all the way there. Proper packet scheduling algorithms must be used to keep real-time data packets moving faster than non-real-time data on the internet. Until then, there is no total guarantee of QoS.

LIST OF REFERENCES

[1]Fredrick Hayes-Roth and Randall Neff. *Specifications for the JTF-ATD Reference Architecture Servers, Version 1.0.* Tecknowledge Federal Systems, March 1995. Written for SAIC at NRaD in San Diego.

[2]Rick Hayes-Roth and Lee Erman. *Joint Task Force Architecture Specification (JTFAS).* Tecknowledge Federal Systems, April 1994. Written for SAIC at NRaD in San Diego.

[3]N. Freed and N. Borenstein. *Multipurpose Internet Mail Extensions, Part One: Format of Internet Message Bodies.* network Working Group of the Internet Engineering Task Force, November 1996. RFC 2045.

[4]Randall Neff and Rick Hayes-Roth. *C2 Schema Lifecycle.* Tecknowledge Federal Systems, September 1996. Written for SAIC at NRaD in San Diego.

[5]Randall Neff and Rick Hayes-Roth. *JTF ATD Implementation Guidlines - V3.2.* Tecknowledge Federal Systems, January 1997. Written for SAIC at NRaD in San Diego.

[6]Anthony Michel. Communications server, 1996. Written for BBN Systems and Technologies.

[7]Robert Orfali, Dan Harkey, and Jeri Edwards. *The Essential Distributed Objects Survival Guide.* John Wiley and Sons, 1996.

[8]T. Kidd, D. Hensgen, R. Freund, and L. Moore. Smartnet: A Scheduling Framework for Heterogeneous Computing. *Proceedings of the IEEE International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN'96)*, June 1996.

[9]Allan Terry, Terry Barnes, and Rick Hayes-Roth. *JTF ATD Communications Server, Architectual Refinement Through Simulation Experiments.* Tecknowledge Federal Systems, September 1995. Written for SAIC at NRaD in San Diego.

[10]Rick Grehan. BYTE's new benchmarks. *BYTE Magazine*, March 1995.

[11]M. Litzkow, M. Livny, and M.W. Mutka. Condor - A Hunter of Idle Workstations. *Proceedings of the 8th International Conference on Distributed Computing*, June 1988.

[12]Naval Command, Control, and Ocean Surveillance Center, Research, Development, Test and Evaluation Division, Code 422, 53140 Gatchell Road, San Diego, CA 92152-7400. *SmartNet Scheduling Tool v2.6 Users Guide*, June 1996.

[13] H. Schulzrinne, GMD Fokus, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. *Audio-Video Transport Working Group fo the Internet Engineering Task Force (RFC 1889)*, January 1996.

[14] L. Delgrossi and L. Berger. Internet Stream Protocol Version 2 (ST2). *ST2 Working Group of the Internet Engineering Task Force (RFC 1819)*, August 1995.

[15] Domenico Ferrari and Dinesh Verma. A Scheme for Real-Time Channel Establishment in Wide-Area Networks. *IEEE Journal on Selected Areas in Communications*, 1991.

[16] Ed R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource Reservation Protocol (RSVP) - Version 1 Functional Specification. *Internet Engineering Task Force*, October 1996.

[17] Rikk Carey, Chris Marrin, and Gavin Bell. The Virtual Reality Modeling Language (VRML) Version 2.0 Specification. *International Standards Organization/International Electrotechnical Commission (ISO/IEC) draft standard 14772*, August 1996.

[18] Calton Pu, Andrew Black, Crispin Cowan, and Jonathan Walpole. Microlanguages for operating system specialization. *SIGPLAN Workshop on Domain-Specific Languages*, January 1997.

[19] Athanasios Papoulis. *Probability, Random Variables, and Stochastic Processes, 2nd. Ed.* McGraw-Hill, New York, 1984.

[20] Naval Command, Control, and Ocean Surveillance Center, Code 422. *SmartNet Scheduling Tool, v2.6, Users Guide*, June 1996.

[21] Taylor Kidd, Debra Hensgen, Richard Freund, Matt Kussow, and Mark Campbell. Compute characteristics: A useful characterization of job runtimes. In preparation for submission (1997).

[22] H. Schulzrinne, GMD Fokus, Casner S, R. Frederick, and V. Jacobson. *RTP: A Transport Protocol for Real-Time Applications.* Audio-Video Transport Working Group of the Internet Engineering Task Force, January 1996. RFC 1889.

[23] Luca Delgrossi, Christian Halstrick, Dietmar Hehmann, Ralf Guido Herrtwich, Oliver Krone, Jochen Sandvoss, and Carsten Vogt. Media Scaling for Audiovisual Communication with the Heidelberg Transport System. In *Proceedings of ACM Multimedia '93*, pages 99–104, Anaheim, CA, August 1993.

[24] L. Delgrossi and L. Berger. *Internet Stream Protocol Version 2 (ST2).* ST2 Working Group of the Internet Engineering Task Force, August 1995. RFC 1819.

[25] Ed. R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. *Resource Reservation Protocol (RSVP) - Version 1 Functional Specification.* Internet Engineering Task Force, October 1996. Internet Draft.

[26] Domenico Ferrari and Dinesh Verma. A Scheme for Real-Time Channel Establishment in Wide-Area Networks. *IEEE Journal on Selected Areas in Communications*, 1991.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center   2
   8725 John J. Kingman Road., Ste 0944
   Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library   2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, CA 93943-5101

3. Director, Training and Education   1
   MCCDC, Code C46
   1019 Elliot Road
   Quantico, VA 22134-5027

4. Director, Marine Corps Research Center   2
   MCCDC, Code C40RC
   2040 Broadway Street
   Quantico, VA 22134-5107

5. Director, Studies and Analysis Division   1
   MCCDC, Code C45
   3300 Russell Road
   Quantico, VA 22134-5130

6. Marine Corps Representative   1
   Naval Postgraduate School
   Code 037, Bldg. 234, HA-220
   699 Dyer Road
   Monterey, CA 93940

7. Marine Corps Tactical Systems Support Activity   1
   Technical Advisory Branch
   Attn: Maj J.C. Cummiskey
   Box 555171
   Camp Pendelton, CA 92055-5080

8. Debra Hensgen   5
   Naval Postgraduate School
   Code CS/Hd, Computer Sciences Dept.
   833 Dyer Rd.
   Monterey, CA 93943-5118

9.Geoffrey Xie   1
   Naval Postgraduate School
   Code CS/Xi, Computer Sciences Dept.
   833 Dyer Rd.
   Monterey, CA 93943-5118

10.John Falby   1
   Naval Postgraduate School
   Code CS/Fa, Computer Sciences Dept.
   833 Dyer Rd.
   Monterey, CA 93943-5118

11.H.J. Siegel   1
   Purdue University
   Room 325, EE Building
   School of Electrical and Computer Engineering
   1285 Electrical Engineer Building
   West Lafayette, IN 47907-1285

12.Richard Freund, Chief Scientist   1
   Heterogeneous Computing Team
   NCCOSC RDTE Div 4221 Rm 341A
   53118 Gatchell Road
   San Diego, CA 92152-7446

13.Taylor Kidd   1
   Naval Postgraduate School
   Code CS/Kt, Computer Sciences Dept.
   833 Dyer Rd.
   Monterey, CA 93943-5118

14.Viktor Prasanna   1
   University of Southern California
   Department of EE-Systems, EEB 200C
   3740 McClintock Ave.
   Los Angeles, CA 90089-2562

15.John Kresho   2
   128 Kerr Rd
   New Kensington, PA 15068